Final Report for Digital Electronics Technology

Project Report: **FPGA-Based Automotive Tail Light System**

Gallo Andrea
Student ID: 2359271
andrea.gallo13@studio.unibo.it

July 27, 2025

Abstract

This project details the design, implementation, and planned verification of an FPGA-based Automotive Tail Light System. The primary objective was to develop a comprehensive signaling unit replicating standard vehicle rear light operations—including brake lights, turn signals, hazard lights, and reverse indication—and to introduce an extended functionality: an emergency mode featuring an on-screen MM:SS timer displayed on a 7-segment display. The system was designed using Verilog HDL, employing a modular approach with distinct components for clock division, input debouncing, finite state machine (FSM) control, emergency timing, and display driving. Theoretical design involved critical timing calculations and FSM state definition. The methodology anticipates simulation using Icarus Verilog and GTKWave for logic verification, followed by hardware implementation and testing on a Xilinx Nexys 4 DDR (Artix-7) FPGA board. Key innovations include the integrated emergency timer and the robust handling of concurrent signal operations. The project demonstrates proficiency in FPGA design flow and HDL-based digital system development, with direct relevance to modern automotive electronic safety systems.

Contents

1 Introduction

1.1 Background

Digital electronic systems are a cornerstone of modern technology, permeating nearly every aspect of daily life and industry. From smartphones and computers to industrial control systems and medical devices, digital logic offers precision, reliability, and ever-increasing processing power [1]. Their ability to process, store, and transmit information in binary format has revolutionized entire sectors, enabling the development of complex and highly integrated solutions. Particularly in the automotive sector, digital electronics have assumed an increasingly crucial role. Modern vehicles integrate numerous digital systems to enhance safety, efficiency, and comfort for both driver and passengers. These systems include Engine Control Units (ECUs), infotainment systems, Advanced Driver-Assistance Systems (ADAS), and, not least, intelligent lighting systems such as the one developed in this project [2]. The ongoing evolution towards increasingly autonomous and connected vehicles further accentuates this trend, making proficiency in digital design technologies an essential skill for electronic engineers.

1.2 Design Goals

The primary objective of this project is to design and implement a comprehensive FPGA-based Automotive Tail Light System. The system aims to simulate the essential lighting operations of a vehicle's rear end, with a focus on clear signaling for various driving conditions.

The **core functionalities** implemented in this system are:

- **Turn Signals:** Independent activation of left and right turn signal indicators, flashing at approximately 1 Hz to indicate the driver's intention to turn or change lanes.
- **Brake Lights:** Illumination of dedicated left, central and right brake lights when the brake input is asserted.
- **Reverse Light:** Activation of a reverse light when the reverse gear input is engaged, functional independently or in conjunction with other lighting states.
- **Hazard Lights (Four-Way Flashers):** Simultaneous flashing of both left and right turn signals, activated by a dedicated input, to alert other road users to a potential hazard.
- **Combined Operations:** The system is designed to correctly manage concurrent activation of signals, such as braking while a turn signal is active, or engaging reverse while other lights are on.

The **extended functionality**, providing an innovative aspect beyond standard operations, is an:

- **Enhanced Emergency Mode with Timer Display:** Upon activation of a specific emergency input (e.g., simultaneous press of two designated buttons), the system enters an enhanced emergency state. In this mode:
  - Both left and right turn signals (hazard lights) flash continuously.
  - Both main brake lights illuminate steadily.
  - The reverse light remains independently controllable based on its specific input.
  - Crucially, a timer is initiated, counting and displaying elapsed time in minutes and seconds (MM:SS format) on a 7-segment display. This timer provides a visual indication of the duration of the emergency event.

The project also aims to demonstrate proficiency in Verilog HDL for digital circuit design and successful implementation on a Xilinx Nexys FPGA platform, showcasing modular design and resource utilization.

1.3 Technology Choice

For the implementation of the Automotive Tail Light System, a Field-Programmable Gate Array (FPGA) platform was selected over a design based on discrete 74-series Integrated Circuits (ICs). While 74-series ICs offer a fundamental approach to digital logic design, FPGAs provide significant advantages for a project of this nature and complexity.

The key reasons for choosing an FPGA (specifically, a Xilinx Nexys4 DDR board) include:

- **Reconfigurability and Flexibility:** FPGAs allow for rapid prototyping and iterative development. The hardware design, described using a Hardware Description Language (HDL) like Verilog, can be easily modified, recompiled, and re-downloaded onto the device. This is invaluable for debugging, implementing changes, and adding new features, such as the extended emergency timer functionality, without altering physical wiring.
- **Integration Density and Complexity Handling:** FPGAs can implement significantly more complex digital systems on a single chip compared to what would be feasible with a multitude of discrete 74-series ICs. The proposed tail light system, with its state machine, debouncers, clock dividers, and timer logic, benefits greatly from this integration capability.
- **Development Efficiency:** HDL-based design streamlines the development process for complex logic. It allows for a more abstract and modular design approach, enhancing readability and maintainability compared to a large schematic of interconnected discrete components. Simulation tools available for HDL designs also facilitate thorough verification before hardware deployment.
- **Alignment with Modern Design Practices and Learning Objectives:** Utilizing an FPGA aligns with contemporary digital design practices and provides valuable experience in HDL programming, synthesis, and implementation on modern programmable logic devices, which are key learning objectives for this Digital Electronics Technology course.

- **Prior Experience and Familiarity:** A significant factor in this choice was prior experience with FPGA technology and the Verilog HDL, gained through other concurrent coursework. This existing familiarity allows for a more focused effort on the design's specific challenges rather than on learning the foundational aspects of the development tools and language from scratch, leading to a more efficient and potentially more sophisticated project outcome.

Therefore, the FPGA platform offers a more robust, flexible, and educationally relevant environment for realizing the design goals of this project, further leveraged by existing student expertise.

2 Design Methodology

This section outlines the methodology adopted for designing the FPGA-based Automotive Tail Light System, covering both the theoretical design phase and the subsequent hardware implementation details.

2.1 Theoretical Design

The theoretical design phase involved breaking down the system into manageable functional blocks, defining their behavior and interactions, and performing necessary calculations for timing-critical components.

2.1.1 Block Diagram of the System

A top-down design approach was employed, starting with a high-level block diagram to visualize the overall architecture of the Automotive Tail Light System. This diagram illustrates the main functional modules and the data/control signal flow between them. The system, as depicted in Figure 1, is comprised of several key interconnected modules:

- **Clock Management Unit (CMU):** Responsible for generating the necessary clock signals (1 Hz for blinking and timing, 1 kHz for display refresh) from the main 100 MHz system clock. This corresponds to the `clock_divider` module.
- **Input Debouncing Units:** Multiple debouncer instances filter noise from push-button inputs (brake) and switch inputs (reverse gear). They also process inputs for the emergency mode activation (combined BTNU and BTNL after debouncing) to ensure reliable signal detection. These are implemented using the `debouncer` module.
- **Finite State Machine (FSM) Controller:** The core logic unit that interprets user inputs (debounced brake, emergency signal, reverse, turn signal switches, hazard switch) and the 1 Hz blink signal. It determines the appropriate state and output signals for the tail lights and activates the emergency timer. This is the `fsm` module.
- **Emergency Timer Unit:** Manages the timing and counting logic (minutes and seconds) when the system is in emergency mode, driven by the 1 Hz clock. This corresponds to the `emergency_timer` module.
- **Seven-Segment Display Driver:** Converts the binary time values (minutes and seconds) from the emergency timer into signals suitable for driving a 4-digit, 7-segment display, handling digit multiplexing using the 1 kHz clock. This is the `seven_seg_driver` module.

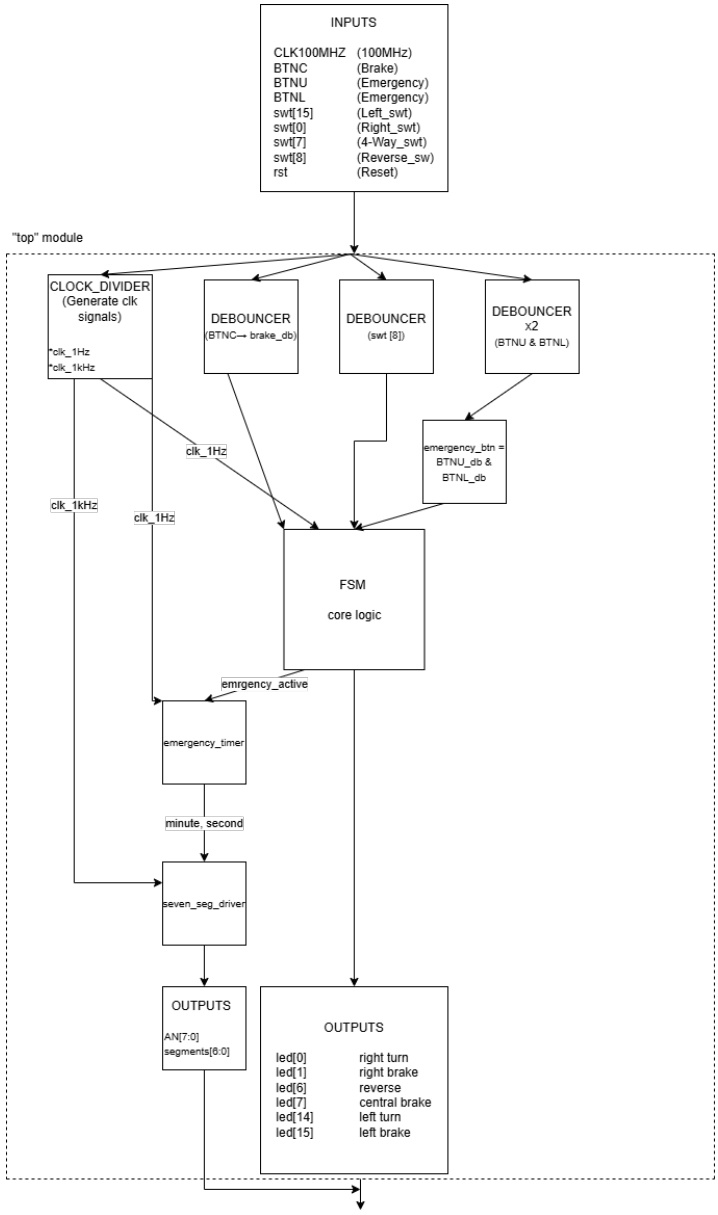

Figure 1: Block Diagram of the FPGA-Based Automotive Tail Light System.

The main inputs to the system include the 100MHz clock, reset signal, buttons for brake and emergency mode activation, and switches for turn signals, hazard, and reverse gear. The primary outputs drive the LEDs for the various light functions and the segments/anodes of the 7-segment display for the emergency timer.

2.1.2 System Control Logic: Finite State Machine (FSM)

The core control logic of the Automotive Tail Light System is implemented as a Finite State Machine (FSM). The FSM is responsible for interpreting various input signals from the driver (simulated via switches and buttons) and transitioning between different operational modes to manage the vehicle's rear lights accordingly. The FSM was designed using Verilog HDL and synthesized for the FPGA.

States Definition: The FSM operates with the following defined states, representing distinct lighting configurations, as illustrated in the state transition diagram (Figure 2):

- `NORMAL`: The default idle state. In this state, turn signals are off. Brake lights activate based on the `brake_btn` input, and the reverse light activates based on the `reverse_sw` input. The emergency timer is inactive (`EmergAct = 0`).
- `LEFT_TURN`: Activated when the `left_sw` is engaged. The left turn indicator flashes, while brake and reverse lights operate based on their respective inputs. The emergency timer is inactive.
- `RIGHT_TURN`: Activated when the `right_sw` is engaged. The right turn indicator flashes, while brake and reverse lights operate based on their respective inputs. The emergency timer is inactive.
- `FOUR_WAY`: Activated when the `four_way_sw` is engaged. Both left and right turn indicators flash simultaneously. Brake and reverse lights operate based on their respective inputs. The emergency timer is inactive.
- `EMERGENCY`: A priority state entered when `emergency_btn` is asserted. In this mode, both turn indicators flash (hazard lights), brake lights are steadily illuminated (ON), and the emergency timer is activated (`EmergAct = 1`). The reverse light remains independently operational based on `reverse_sw`.

State Transitions and Output Logic: State transitions are determined by the current state and the status of input signals: the global reset (`rst`), the emergency activation button (`emergency_btn`), and switches for left turn (`left_sw`), right turn (`right_sw`), and four-way flashers (`four_way_sw`). The 1 Hz `blink` signal (derived from the clock divider) enables the flashing behavior of turn signals. The detailed state transition diagram, illustrating the states, transition conditions, and key outputs/actions within each state, is presented in Figure 2.

The global `rst` signal forces the FSM to the `NORMAL` state from any other state. The `emergency_btn` acts as a toggle: if asserted, the FSM enters the `EMERGENCY` state (or transitions to `NORMAL` if already in `EMERGENCY` and the button is pressed again). If not in `EMERGENCY` mode, transitions from `LEFT_TURN`, `RIGHT_TURN`, or `FOUR_WAY` back to `NORMAL` occur when their respective activating switch is de-asserted, with conditions to prevent unintended transitions (e.g., if `four_way_sw` remains active). The output logic, primarily dependent on the current state (Moore-type behavior for major light configurations) but also influenced by direct inputs like `brake_btn` and `reverse_sw` for concurrent operations, controls the `leds` bus and the `emergency_active` signal.

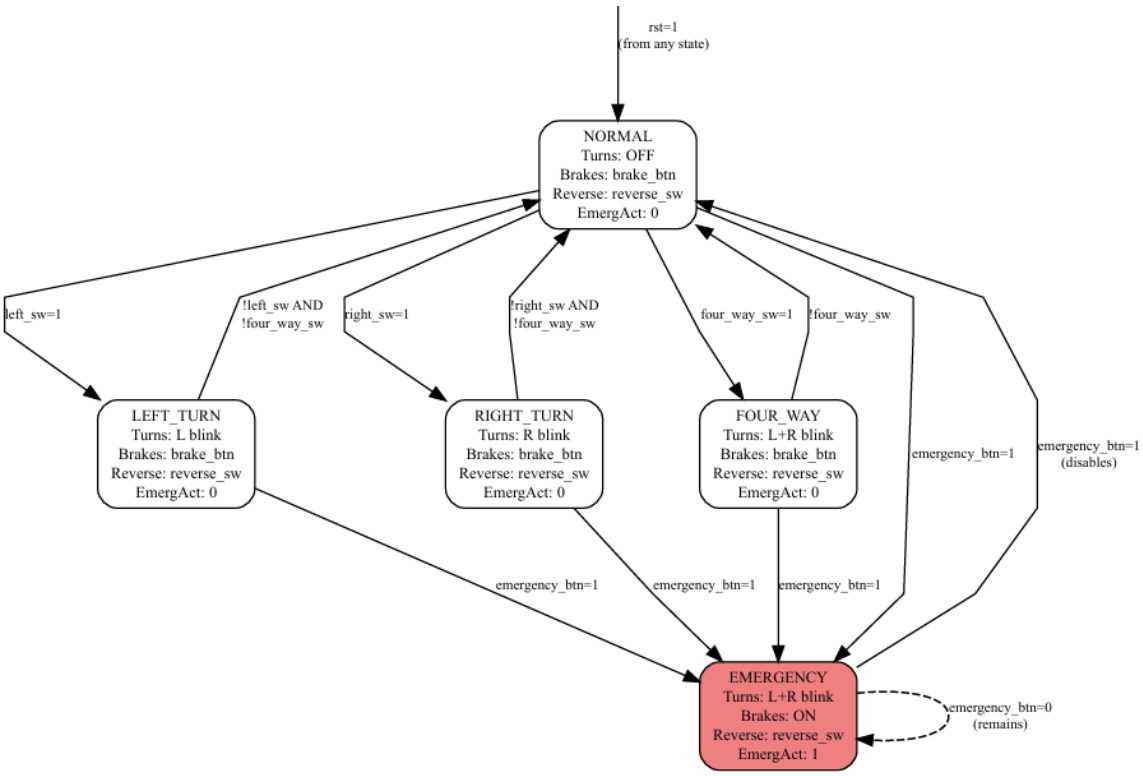

Figure 2: State Transition Diagram for the Automotive Tail Light FSM.

2.1.3 Critical Calculations and Parameter Settings

Several critical calculations were performed during the design phase to ensure correct timing for various system operations, particularly for clock division, input debouncing, and the emergency timer functionality. These calculations are based on fundamental principles of digital timing and counter design, as detailed in standard digital electronics literature [1] and even during this course. The system operates with a primary input clock of 100 MHz ($f_{sys\_clk} = 100$ MHz).

Clock Divider Calculations: The `clock_divider` module generates two essential clock signals from the 100 MHz system clock. The principle involves using a counter to divide the high-frequency system clock down to the desired lower frequencies. The relationship between the clock period, clock frequency, and the required counter limit forms the basis of this calculation [1, Chapters on Counters and Timers].

- **1 Hz Clock (clk_{1Hz}):** This clock is used for the blinking rate of turn/hazard signals and as the time base for the `emergency_timer`. To generate a 1 Hz signal (period $T_{1Hz} = 1s$), the output should toggle every 0.5 seconds. The number of system clock cycles required for one toggle (half period) is derived as:

$$N_{1Hz\_toggle} = \frac{T_{1Hz}/2}{T_{sys\_clk}} = \frac{0.5 \text{ s}}{1/(100 \times 10^6 \text{ Hz})} = 0.5 \times 100 \times 10^6 = 50,000,000$$

  The counter in the Verilog module (`counter_1Hz`) therefore counts from 0 up to $N_{1Hz\_toggle} - 1 = 49,999,999$. The register `counter_1Hz` is defined as `reg [26:0]`, which can hold values up to $2^{27} - 1$, sufficient for this count.

- **1 kHz Clock (clk_{1kHz}):** This clock is used for refreshing the 7-segment display to avoid flickering. To generate a 1 kHz signal (period $T_{1kHz} = 1ms$), the output should toggle every 0.5 ms. The number of system clock cycles required for one toggle is:

$$N_{1kHz\_toggle} = \frac{T_{1kHz}/2}{T_{sys\_clk}} = \frac{0.5 \times 10^{-3} \text{ s}}{1/(100 \times 10^6 \text{ Hz})} = 0.5 \times 10^{-3} \times 100 \times 10^6 = 50,000$$

  The counter in the Verilog module (`counter_1kHz`) counts from 0 up to $N_{1kHz\_toggle} - 1 = 49,999$. The register `counter_1kHz` is defined as `reg [16:0]`, which can hold values up to $2^{17} - 1$, sufficient for this count.

Debouncer Timing: The `debouncer` module is designed to filter out mechanical bounces from push-button and switch inputs. This is achieved by ensuring an input signal remains stable for a minimum period before its state change is recognized. The duration of this period is determined by a counter clocked by the system clock [1, Chapter on Interfacing and Input Conditioning]. A stable output is produced after the input signal remains constant for a predetermined duration. The counter within the debouncer module (`counter`) is `reg [20:0]`. The debounce period is set by the counter reaching a value of $2,000,000$:

$$T_{debounce} = \text{Counter\_Limit} \times T_{sys\_clk} = 2,000,000 \times \frac{1}{100 \times 10^6 \text{ Hz}} = 2,000,000 \times 10 \text{ ns} = 20 \text{ ms}$$

This 20 ms debounce time is a common value, generally effective for most mechanical switches and buttons.

Emergency Timer Limits: The `emergency_timer` module counts and displays time in minutes and seconds. The design of this timer involves standard counter logic for seconds (modulo-60) and minutes (modulo-16 in this implementation), concepts also covered in [1]:

- **Seconds Counter (`seconds`):** This is a 6-bit register (`reg [5:0]`) counting from 0 to 59.
- **Minutes Counter (`minutes`):** This is a 4-bit register (`reg [3:0]`) counting from 0 to 15. The maximum displayable time for the emergency event is therefore 15 minutes and 59 seconds.

These limits were chosen to provide a reasonable duration for the extended feature while fitting within a 4-digit display (MM:SS).

2.2 Hardware/FPGA Implementation

This section details the implementation of the Automotive Tail Light System on the target FPGA platform, including resource utilization and key aspects of the Verilog code. The design was synthesized and implemented using Xilinx Vivado Design Suite for the **Xilinx Nexys4 DDR board, which features an Artix-7 XC7A100TCSG324-1 FPGA**.

2.2.1 FPGA Resource Utilization

Following the synthesis and implementation stages using the Xilinx Vivado Design Suite, the resource utilization of the design on the Artix-7 XC7A100TCSG324-1 FPGA (as present on the Nexys 4 DDR) was analyzed. The system was found to be efficiently implemented, utilizing a small fraction of the available resources. This indicates that the design is well-optimized and leaves ample room for potential future expansions or integration with other functionalities.

A summary of the key resources utilized, as reported by Vivado, is presented in Figure 3.

Name 1	Slice LUTs (63400)	Bonded IOB (210)	BUFGCTRL (32)	Slice Registers (126800)	F7 Muxes (31700)	Slice (15850)	LUT as Logic (63400)
top	114	40	1	171	5	76	114
clk_div (clock_divider)	42	0	0		0	25	42
deb_brake (debouncer)	7	0	0		0	10	7
deb_l (debouncer_0)	7	0	0		0	10	7
deb_reverse (debouncer_1)	7	0	0		0	10	7
deb_u (debouncer_2)	8	0	0		0	9	8
display (seven_seg_driver)	26	0	0		5	10	26
fsm_inst (fsm)	9	0	0		0	4	9
timer (emergency_timer)	8	0	0		0	4	8

Figure 3: Vivado Resource Utilization Report Summary for the Automotive Tail Light System on the Artix-7 XC7A100TCSG324-1 FPGA.

The detailed resource utilization for the entire design (‘top‘ module) and its constituent submodules, as reported by the Xilinx Vivado synthesis tool, is presented in Figure 3. The data clearly demonstrates the design's efficiency and modest footprint on the Artix-7 FPGA.

For the top-level design, the key utilized resources are:

- **Slice LUTs:** 114 out of 63,400 available (approximately 0.18%). These Look-Up Tables implement the combinatorial logic of the system.
- **Slice Registers:** 171 out of 126,800 available (approximately 0.13%). These are primarily the flip-flops used for storing state in the FSM, counters, and debouncer circuits.
- **Bonded IOBs (Input/Output Buffers):** 40 out of 210 available (approximately 19.05%). This count directly reflects the number of physical FPGA pins used for the system clock, reset, push-buttons, slide switches, LEDs, and the 7-segment display, as specified in the XDC constraints file (see Appendix A.2).
- **BUFGCTRL (Global Clock Buffers):** 1 out of 32 available (approximately 3.13%). A single global clock buffer is utilized to ensure proper distribution of the main 100 MHz clock signal throughout the FPGA fabric, minimizing skew and ensuring reliable synchronous operation.
- **F7 Muxes:** 5 out of 31,700 available (a negligible percentage). These are specific multiplexer resources within the FPGA slices.
- **Slices:** 76 out of 15,850 available (approximately 0.48%). This represents the number of fundamental FPGA logic slices occupied by the design.

The utilization figures for Slice LUTs, Slice Registers, and overall Slices are exceptionally low, underscoring the compactness of the Verilog HDL design and its efficient mapping to the FPGA architecture. For a control-logic-dominant project, specialized resources like Block RAM (BRAM) or DSP Slices were not required and thus show no utilization. The breakdown by module in Figure 3 also allows for an analysis of resource consumption by each functional block, with the `clk_div` module understandably contributing a significant portion of the LUTs due to its large counters, and the `display` module using a notable number of LUTs and some F7 Muxes for the 7-segment driving logic.

The overall low resource usage confirms that the design is well within the capabilities of the target Artix-7 device, leaving substantial room for future expansions or the integration of more complex functionalities.

2.2.2 RTL Schematic

The Xilinx Vivado Design Suite was used to synthesize the Verilog HDL code into a netlist targeting the Artix-7 FPGA. After synthesis, an RTL (Register Transfer Level) schematic can be generated, which provides a graphical representation of the design's architecture in terms of interconnected registers, multiplexers, arithmetic units, and other logic blocks inferred from the HDL description.

Figure 4 presents the top-level RTL schematic of the Automotive Tail Light System. This diagram illustrates the instantiation of all major Verilog modules: `clock_divider`, multiple `debouncer` instances for input conditioning, the `fsm` (Finite State Machine) core logic, the `emergency_timer`, and the `seven_seg_driver`. The connections between these modules, representing data paths and control signals, are clearly visible, as are some automatically inferred logic gates (e.g., for combining debounced button signals to create the emergency button input, or for driving specific LED outputs). This schematic view is invaluable for visualizing the synthesized hardware structure and verifying high-level connectivity before proceeding to implementation and bitstream generation.

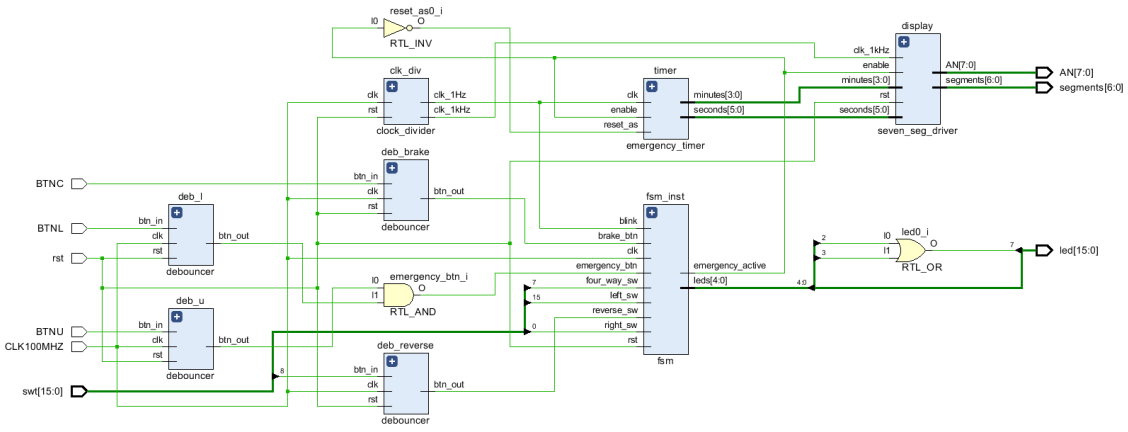

Figure 4: Top-Level RTL Schematic of the Automotive Tail Light System generated by Vivado.

2.2.3 Key Verilog Code Snippets and Annotations

The Automotive Tail Light System was implemented using Verilog HDL, following a modular design approach. This section highlights key code segments from different modules to illustrate the design logic and implementation techniques. The complete Verilog source code is provided in Appendix A.1.

Finite State Machine Module (`fsm.v`): The `fsm.v` module's behavior is primarily governed by its state transitioning mechanism. It employs `localparam` declarations to define the five principal operational states: `NORMAL`, `LEFT_TURN`, `RIGHT_TURN`, `FOUR_WAY`, and `EMERGENCY`, using an explicit binary encoding. The current state of the machine is held in the `current_state` register, which is updated synchronously on each positive clock edge based on the value of `next_state`. A global reset signal (`rst`) initializes the FSM to the `NORMAL` state.

Listing 1: FSM State Definition and Register Logic (`fsm.v`)

```
    // State definitions (Explicit binary encoding)
    localparam [2:0]
     EMERGENCY  = 3'd0,
     LEFT_TURN  = 3'd1,
     RIGHT_TURN = 3'd2,
     FOUR_WAY   = 3'd3,
     NORMAL     = 3'd4;

    reg [2:0] current_state, next_state;

    // Internal signal for turn indicator LEDs {left, right}
    reg [1:0] turn_leds;

    // Sequential logic for state register
    always @(posedge clk) begin
        if (rst) current_state <= NORMAL;
        else current_state <= next_state;
    end
```

The combinational logic for determining the `next_state` (Listing 2) evaluates the `current_state` along with various input signals. The `emergency_btn` input is given the highest priority; its assertion toggles the FSM into or out of the `EMERGENCY` state. Otherwise, if not in `EMERGENCY`, the `next_state` is determined by a `case` statement based on `left_sw`, `right_sw`, and `four_way_sw`, including conditions for returning to `NORMAL`.

Listing 2: FSM Next-State Combinational Logic (`fsm.v`)

```
    // Combinational logic for state transitions
    always @(*) begin
        next_state = current_state; // Default: remain in current state

        // Emergency button toggles EMERGENCY state and has highest priority
        if (emergency_btn) begin
            next_state = (current_state == EMERGENCY) ? NORMAL : EMERGENCY;
        end
        // Logic for other states if not in or transitioning to/from EMERGENCY
        else if (current_state != EMERGENCY) begin
            case (current_state)
                NORMAL: begin // From NORMAL, can go to turn signals or four-way
                    if (four_way_sw)       next_state = FOUR_WAY;
                    else if (left_sw)      next_state = LEFT_TURN;
                    else if (right_sw)     next_state = RIGHT_TURN;
                end
                // Return to NORMAL if respective switch is turned off
                LEFT_TURN:  if (!left_sw && !four_way_sw)   next_state = NORMAL;
                RIGHT_TURN: if (!right_sw && !four_way_sw)  next_state = NORMAL;
                FOUR_WAY:   if (!four_way_sw)               next_state = NORMAL;
            endcase
        end
    end
```

The FSM's output logic, also combinational, generates the 5-bit `leds` vector and the `emergency_active` signal. The internal mapping for the `leds` output bus within the FSM is defined as: `leds[4]` for reverse, `leds[3]` for left brake, `leds[2]` for right brake, `leds[1]` for left turn, and `leds[0]` for right turn. An internal 2-bit signal, `turn_leds`, manages the blinking logic based on the `current_state` and the 1 Hz `blink` input. The final assignment to `leds` and `emergency_active` depends on whether the FSM is in the `EMERGENCY` state. In `EMERGENCY`, `emergency_active` is high, turn signals blink, and brake lights are forced ON. In other states, `emergency_active` is low, turn signals operate as per their state, and brake lights depend on `brake_btn`. The reverse light (`leds[4]`) is always controlled by `reverse_sw`. This structure ensures correct lighting for all modes with priority for emergency conditions. The full output logic is detailed in Appendix A.1.

Emergency Timer (`emergency_timer.v`): The `emergency_timer` module (Listing 3) tracks elapsed time in minutes (0-15) and seconds (0-59) during an emergency. It operates on a 1 Hz clock (`clk`) and is controlled by an `enable` signal (asserted when `emergency_active` is high) and a reset signal. A key aspect of this module is the handling of its reset input (`reset_as`), which is asynchronous to the timer's clock. In this implementation, a **single-flop synchronizer** is used: the `reset_as` signal is registered once with the timer's `clk` to produce `reset_sync`. This `reset_sync` signal, or the de-assertion of the `enable` signal, will reset the timer's `minutes` and `seconds` counters to zero. When enabled and not in reset, the timer increments seconds, rolling over to increment minutes as appropriate. This approach to reset synchronization with a single flip-flop is chosen for its simplicity in this context, though for critical cross-domain signals, a two-flop synchronizer is generally preferred to further minimize metastability risks.

Listing 3: Verilog code for the `emergency_timer` module (revised)

```
// Timer to count minutes and seconds, active during emergency mode
module emergency_timer (
    input clk,              // Clocked by 1Hz for second counting
    input reset_as,         // Asynchronous reset (active high to reset timer)
    input enable,           // Enables counting when emergency is active
    output reg [3:0] minutes,
    output reg [5:0] seconds
);
    // Single-flop synchronizer for asynchronous reset
    reg reset_sync;
    always @(posedge clk) reset_sync <= reset_as;

    // Counter logic for seconds and minutes (up to 15:59)
    always @(posedge clk) begin
        if (reset_sync || !enable) begin // Reset if reset asserted or not enabled
            minutes <= 0;
            seconds <= 0;
        end
        else if (enable) begin // Count when enabled
            if (seconds == 59) begin
                seconds <= 0;
                minutes <= (minutes == 15) ? 0 : minutes + 1;
            end
            else seconds <= seconds + 1;
        end
    end
endmodule
```

Input Debouncing Logic (`debouncer.v`): To ensure reliable operation from physical switches and buttons, which are prone to contact bounce, a `debouncer` module is utilized (Listing 4). This module validates a signal transition only if the new signal level remains stable for a predetermined period (20 ms in this design, with a 100 MHz clock), effectively filtering out spurious transients. The logic involves a counter that starts when the input `btn_in` differs from the stable output `btn_out`. The output is updated only if the input remains consistently different for the full debounce duration. This strategy is applied to inputs like the brake button and reverse switch. (Refer to Section 2.1.3 for calculation details).

Listing 4: Verilog code for the `debouncer` module (revised)

```
// Module to debounce a button input, ensuring a stable signal
module debouncer (
    input clk,
    input rst,
    input btn_in,
    output reg btn_out
);
    // Counter to measure input stability duration
    reg [20:0] counter;

    always @(posedge clk) begin
        if (rst) begin
            counter <= 0;
            btn_out <= 0;
        end
        // If input changes, start/continue counting.
        // Update output only after the input has been stable for a defined period.
        else if (btn_in != btn_out) begin
            counter <= counter + 1;
            if (counter == 21'd2_000_000)  btn_out <= btn_in; // Threshold
        end
        // If input is stable or returns to previous state, reset counter.
        else counter <= 0;
```

```
    end
endmodule
```

3 Results & Analysis

This section presents the results obtained from simulation and hardware testing of the FPGA-based Automotive Tail Light System, along with an analysis of its performance.

3.1 Simulation

Prior to hardware implementation, a comprehensive Verilog testbench was developed to rigorously verify the logical correctness of the core modules: the Finite State Machine (`fsm`), the `emergency_timer`, and the `seven_seg_driver`. The testbench, executed using Icarus Verilog, applied a structured sequence of input stimuli to simulate various operational scenarios. For efficiency, the 1 Hz blink signal and the 1 kHz display refresh clock were accelerated within the testbench environment. Simulation outputs were captured in VCD (Value Change Dump) files and visualized using GTKWave. The testbench also incorporated `$display` system tasks to provide textual feedback on the status of individual test cases.

The simulations aimed to verify several key aspects of the design:

- Correct FSM state transitions in response to brake, turn signal, hazard, reverse, and emergency inputs.
- Proper output assertion by the FSM for all lighting configurations, including concurrent operations.
- Accurate counting (seconds and minutes) and reset behavior of the `emergency_timer` module, triggered by the FSM's emergency status.
- Correct digit multiplexing and segment data generation by the `seven_seg_driver` for the 7-segment display.
- Overall system coherence and correct interaction between the instantiated modules.

The following figures present selected waveform captures from these simulations, illustrating critical functionalities. The full testbench code is provided in Appendix B.

FSM Normal Operations Verification

Figure 5 illustrates the FSM's response to several standard operational inputs. The simulation demonstrates the system's behavior during reset, activation of the brake signal (`brake_btn`), engagement of the left turn signal (`left_sw`), and activation of the four-way hazard lights (`four_way_sw`). The `leds_fsm` output bus correctly reflects the state of the corresponding lights, with turn signals and hazard lights blinking in sync with the `blink` signal.

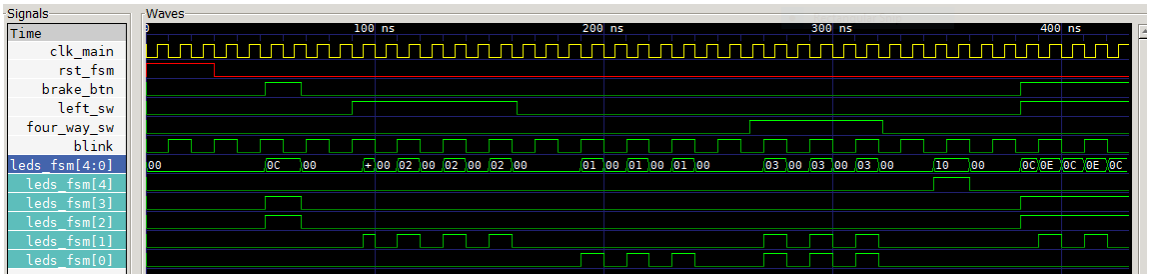

Figure 5: Simulation waveform showing FSM response to reset, brake, left turn, and hazard signals. Key signals: `clk_main`, `rst_fsm`, `brake_btn`, `left_sw`, `four_way_sw`, `blink`, `leds_fsm[4:0]`.

FSM Emergency Mode and Timer Activation

The activation of the emergency mode is a critical feature. Figure 6 shows the FSM transitioning into the `EMERGENCY` state. Observe the `emergency_active` signal going high, which in turn enables the `emergency_timer`. The `leds_fsm` output shows brake lights on steadily and hazard lights blinking. The test also demonstrates exiting the emergency mode.

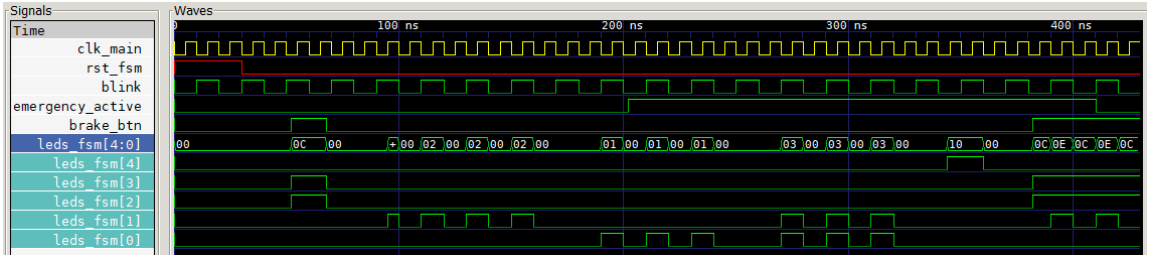

Figure 6: Simulation waveform demonstrating FSM entry into and exit from emergency mode. Key signals: `clk_main`, `rst_fsm`, `emergency_active`, `leds_fsm[4:0]`.

Emergency Timer Operation

Figure 7 details the behavior of the `emergency_timer`. Once `emergency_active` is asserted (forced in this specific test segment or activated by the FSM), the timer begins counting seconds and minutes, clocked by the (accelerated) 1 Hz `blink` signal. The waveform shows the `seconds` counter rolling over at 59 to increment the `minutes` counter. The timer also correctly resets to zero when `emergency_active` is de-asserted.

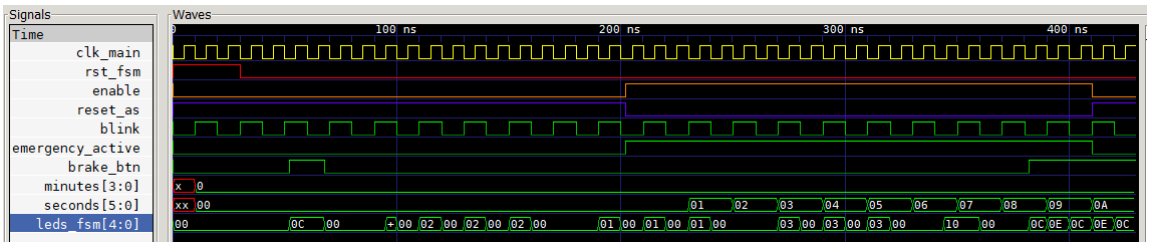

Figure 7: Simulation waveform illustrating the `emergency_timer` counting seconds and minutes. Key signals: `blink` (as timer clock), `emergency_active` (as enable/reset), `minutes[3:0]`, `seconds[5:0]`.

Seven-Segment Display Driver Multiplexing

The correct operation of the 7-segment display driver is crucial for visualizing the emergency timer. Figure 8 demonstrates the multiplexing action of the `seven_seg_driver`. The `AN[7:0]` signals cycle through, activating one display digit at a time, synchronized with the (accelerated) 1 kHz `clk_disp` signal.

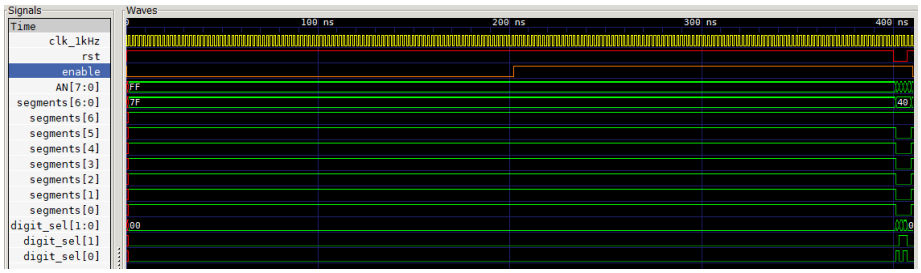

Figure 8: Simulation waveform showing the multiplexing of anode signals (`AN`) by the `seven_seg_driver`. Key signals: `clk_disp`, `rst_disp`, `AN[7:0]`

The simulation results confirmed that all core logic modules and their integration performed as expected under the defined test scenarios, providing confidence for subsequent hardware implementation.

3.2 Hardware Testing

Following the Verilog design phase, the Automotive Tail Light System was synthesized and implemented on the Xilinx Nexys 4 DDR development board, which features an Artix-7 FPGA (XC7A100TCSG324-1). This section details the hardware testing setup, procedures, observed results, resource utilization, and performance metrics obtained from Xilinx Vivado IDE.

3.2.1 Testing Setup and Procedure:

The hardware testing was conducted entirely using the on-board peripherals of the Nexys 4 DDR board. Inputs to the system were manipulated as follows:

- The global reset (`rst`) was triggered using the `BTNR` push-button.
- The brake signal (`BTNC`), and the buttons for emergency mode activation (`BTNU`, `BTNL`) were actuated via their respective on-board push-buttons.
- Turn signal engagement (left: `swt[15]`, right: `swt[0]`), hazard light activation (`swt[7]`), and reverse gear selection (`swt[8]`) were controlled using the on-board slide switches.

System outputs were observed visually through:

- The on-board discrete LEDs (`led[0]`, `led[1]`, `led[6]`, `led[7]`, `led[14]`, `led[15]`) indicating the status of individual light functions (brakes, turn signals, reverse).
- The on-board four-digit seven-segment display, which showed the elapsed time (MM:SS) for the emergency timer feature.

A systematic testing procedure was employed. Initially, each function was tested in isolation (e.g., activating only the left turn signal, then deactivating it; testing brake lights independently, etc.). Subsequently, combined scenarios were tested to verify correct concurrent operation (e.g., activating a turn signal, then engaging the brakes, then disengaging brakes, then deactivating the turn signal). All FSM state transitions and the full operation of the emergency timer were verified.

3.2.2 Observed Results and Functionality Verification:

The hardware prototype performed flawlessly and in complete accordance with the design specifications. Specific observations include:

- All turn signals (left, right, and hazard/four-way) blinked clearly at the designed 1 Hz rate.
- Brake lights illuminated correctly and promptly upon assertion of the brake input, functioning correctly both independently and concurrently with active turn signals or reverse light.
- The reverse light activated as expected when the corresponding switch was engaged.
- The emergency mode was successfully initiated by the simultaneous press of BTNU and BTNL. This correctly activated flashing hazard lights, steady brake lights, and the emergency timer on the 7-segment display. The timer counted accurately in minutes and seconds and was correctly reset upon exiting the emergency mode.
- The 7-segment display provided a clear, flicker-free readout of the timer values.

- All state transitions within the FSM were observed to be smooth and corresponded precisely to the intended logic, with no unexpected behavior or glitches during the manual operation of inputs.

The consistent and correct visual feedback from the LEDs and the 7-segment display robustly confirmed the logical correctness of the implemented Verilog code on the physical hardware.

3.2.3 Visual Documentation of Working Prototype

The operational correctness of the system on the FPGA board was visually confirmed. Figures 9 and 10 provide photographic evidence of the system functioning in different key states.

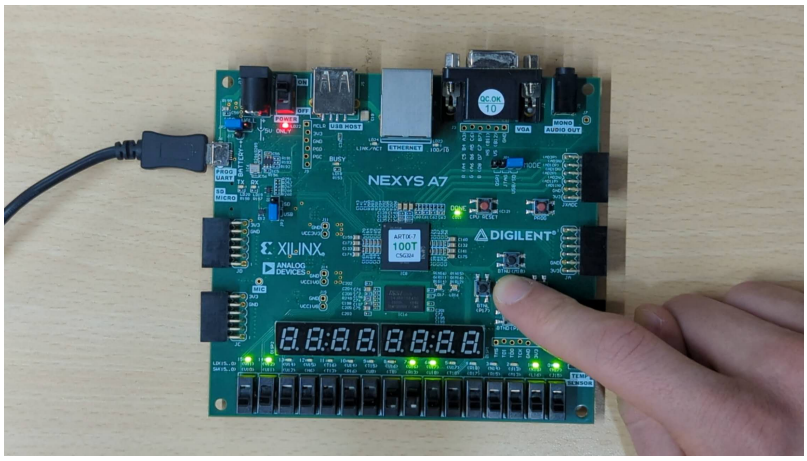

Figure 9: Hardware prototype in operation: Demonstrating simultaneous activation of hazard lights (both turn indicator LEDs blinking), brake lights (brake indicator LEDs ON), and the reverse light (reverse indicator LED ON).

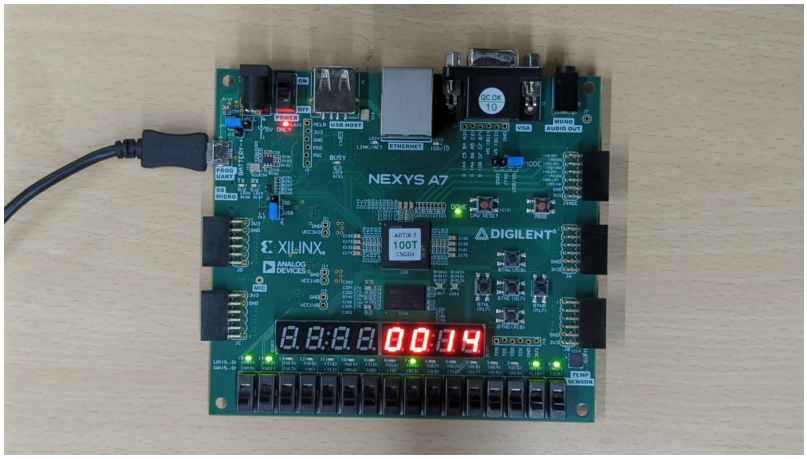

Figure 10: Hardware prototype in emergency mode: Hazard lights flashing, brake lights steadily illuminated, and the emergency timer actively counting and displaying time on the 7-segment display.

3.2.4 Timing Performance Analysis

The timing performance of the implemented design was analyzed using the static timing analysis reports generated by the Xilinx Vivado Design Suite after place and route. These reports are crucial for verifying that the design meets all setup, hold, and pulse width timing requirements for the specified 100 MHz system clock (`CLK100MHZ`).

Figure 11 presents a snapshot of the key timing metrics from the Vivado timing summary report.

Clock	Edges (WNS)	WNS (ns)	TNS (ns)	Failing Endpoints (TNS)	Total Endpoints (TNS)	Edges (WHS)	WHS (ns)	THS (ns)	Failing Endpoints (THS)	Total Endpoints (THS)	WPWS (ns)	TPWS (ns)	Failing Endpoints (TPWS)	Total Endpoints (TPWS)
sys_clk_pin	rise - rise	6.207	0.000	0	221	rise - rise	0.228	0.000	0	221	4.500	0.000	0	138

Figure 11: Summary of Post-Implementation Timing Analysis from Vivado for the `sys_clk_pin`.

As shown in Figure 11, the critical timing metrics for the `sys_clk_pin` are:

- Worst Negative Slack (WNS) for setup time: **6.207 ns**.
- Total Negative Slack (TNS) for setup time: **0.000 ns**.
- Worst Hold Slack (WHS) for hold time: **0.228 ns**.
- Total Hold Slack (THS) for hold time: **0.000 ns**.
- Worst Pulse Width Slack (WPWS) for pulse width: **4.500 ns**.

All reported slack values are positive, indicating that there are no setup, hold, or pulse width violations. This confirms a robust timing closure and reliable operation of the design at the 100 MHz target frequency.

3.2.5 Power Consumption Estimation

An estimation of the on-chip power consumption was obtained using the power analysis tools within the Xilinx Vivado Design Suite. Figure 12 shows the summary of this power estimation.

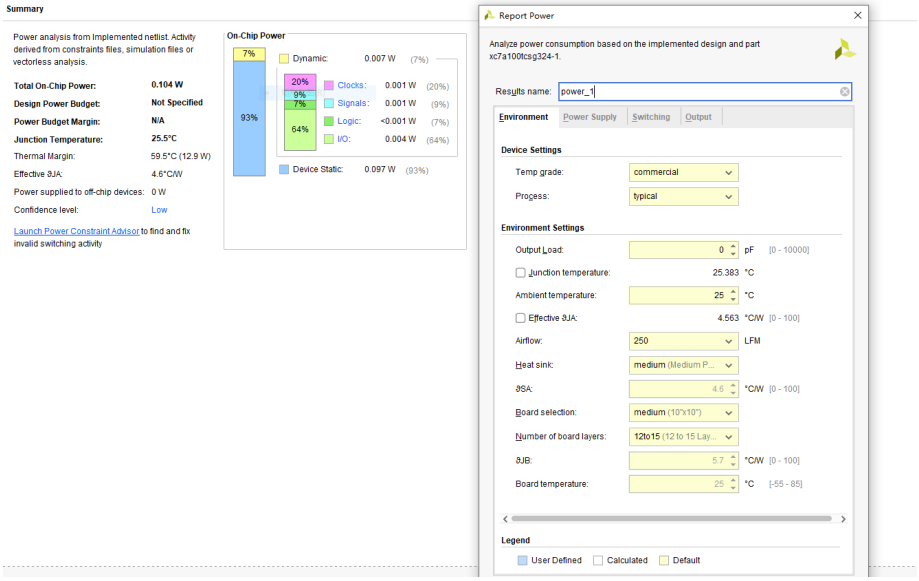

Figure 12: Summary of Power Consumption Estimation from Vivado for the Automotive Tail Light System.

According to the Vivado power report, summarized in Figure 12, the total estimated on-chip power consumption is approximately **0.104 W (104 mW)**. This is composed of:

- Dynamic Power: **0.007 W (7 mW)** (7% of total on-chip power).
- Device Static Power: **0.097 W (97 mW)** (93% of total on-chip power).

The report indicates a "Low" confidence level for this estimation, primarily due to the absence of detailed simulation activity files (e.g., SAIF) to accurately model signal switching rates. The settings used for the analysis (e.g., ambient temperature 25°C, typical process) resulted in an estimated junction temperature of 25.5°C. Despite the low confidence in absolute dynamic power values, the report provides a useful baseline for the design's power characteristics, highlighting that static power is the dominant component for this Artix-7 device under these conditions.

3.3 Comparison of Simulation and Hardware Results

A crucial step in the validation of any FPGA design is the comparison between the behavior observed during simulation and the performance of the actual hardware implementation. This ensures that the simulation environment accurately modeled the system and that the synthesis and implementation processes did not introduce unintended discrepancies.

In this project, the extensive simulations performed using Icarus Verilog and visualized with GTKWave (detailed in Section 3.1) served as the primary means of verifying the logical correctness of the Automotive Tail Light System prior to hardware deployment. The testbench was designed to cover a wide range of operational scenarios, including individual light functions, concurrent operations, FSM state transitions, and the functionality of the emergency timer and display driver.

The subsequent hardware testing on the Xilinx Nexys 4 DDR board (detailed in Section 3.2) provided a direct means to validate these simulation results in a real-world environment. The observed behavior of the physical prototype, including the responses of the on-board LEDs and the 7-segment display to stimuli from switches and push-buttons, was meticulously compared against the expected outcomes derived from the simulation waveforms.

The hardware tests demonstrated a **strong correlation** with the simulation results. All functionalities, such as the correct blinking rate of turn/hazard signals, the prompt activation of brake and reverse lights, the accurate FSM state transitions, the precise counting of the emergency timer, and the clear multiplexed display of timer values, performed on the FPGA board exactly as predicted by the simulations. There were no observable discrepancies between the simulated logic and the hardware behavior. This congruence confirms that the Verilog design was robust, the testbench provided adequate coverage, and the synthesis/implementation process faithfully translated the HDL description into a functional hardware circuit.

Furthermore, the final hardware implementation successfully met all the core and extended design objectives outlined in Section 1.2. The system effectively replicates standard automotive tail light operations and incorporates the specified innovative feature of an emergency timer with a visual display, thereby fulfilling the project's primary goals. The positive timing analysis results (Section 3.2.4) also confirm that the design operates reliably at the target 100 MHz system clock frequency, a prerequisite for correct functional behavior.

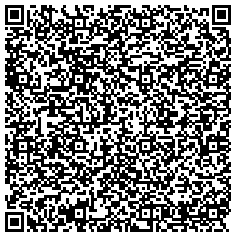

Figure 13: QR Code for accessing the hardware demonstration video, illustrating the functionalities discussed in Section 1.2

4 Discussion

This section reflects on the design and development process of the FPGA-based Automotive Tail Light System. It discusses the challenges encountered, optimization strategies employed, and potential limitations of the current design, along with suggestions for future improvements.

4.1 Challenges Encountered

Several challenges were encountered during the design and implementation of this project, requiring iterative refinement and careful debugging:

- **FSM Logic Design and Refinement:** Developing the core Finite State Machine (FSM) logic proved to be a significant challenge. The initial design underwent several iterations to correctly manage all specified operational states (Normal, Left/Right Turn, Four-Way, Emergency) and accurately handle the numerous transitions triggered by multiple asynchronous inputs. Ensuring the correct priority for signals, such as the emergency button, and managing concurrent operations (e.g., braking while a turn signal is active) required careful logical structuring and extensive simulation. Arriving at the final, robust FSM logic was a laborious but critical process.
- **Testbench Development Strategy:** Crafting effective Verilog testbenches to thoroughly verify the functionality of individual modules and the integrated system was a laborious job. Determining the appropriate stimuli sequences to cover all relevant operational scenarios, edge cases, and timing interactions to ensure comprehensive code coverage was a key difficulty that was overcome through iterative development of the test environment.
- **Verilog Syntax and Debugging Nuances:** Minor typographical errors or subtle misunderstandings of Verilog syntax occasionally led to simulation or synthesis failures that were not immediately obvious. Locating these small mistakes within the codebase sometimes proved to be time-consuming, emphasizing the need for meticulous coding practices and careful review of compiler/simulator error messages.
- **Risk of Metastability with Asynchronous Reset Signals:** A significant design consideration for the `emergency_timer` module was managing its asynchronous reset input (`reset_as`), derived from the `~emergency_active` signal. When an asynchronous signal directly controls synchronous logic, such as the counters within the timer clocked by `clk_1Hz`, there is an inherent risk of inducing a metastable state in the flip-flops of that synchronous logic if the asynchronous signal transition violates setup or hold times relative to the clock. Metastability can lead to unpredictable system behavior, where outputs may oscillate or settle to an incorrect value for an indeterminate period [3, Chapter 3.5.4]. Recognizing this potential issue was crucial for ensuring the reliable operation of the timer's reset mechanism.

Overcoming these challenges was instrumental in developing a functional and robust design.

4.2 Optimization Strategies Applied

Several design strategies were employed throughout the development of the Automotive Tail Light System to enhance its robustness, reliability, and maintainability. These include:

- **Input Signal Debouncing:** To counteract the inherent mechanical bouncing of physical switches and push-buttons, dedicated `debouncer` modules were implemented for all critical discrete inputs (brake, emergency activation buttons, reverse switch). As detailed in Section 2.1.3, these modules introduce a 20 ms delay, ensuring that only stable signal transitions are propagated to the core logic, thereby preventing erroneous FSM transitions or unintended behavior. This significantly improves the system's reliability in a physical hardware environment.

- **Synchronization of Asynchronous Timer Reset:** To address the potential metastability risk associated with the asynchronous reset of the `emergency_timer` (as discussed in Section 4.1), a specific synchronization strategy was implemented. The asynchronous reset signal `reset_as` was passed through a single D flip-flop (`reset_sync <= reset_as;`) clocked by the timer's own `clk_1Hz` before being used by the timer's internal logic. This technique aligns the reset signal to the timer's clock domain, thereby providing a synchronized version (`reset_sync`) for the synchronous reset logic. While multi-stage synchronizers are often preferred for maximum robustness, this single-stage approach serves as a fundamental optimization to improve the signal integrity and reliability of the reset operation compared to using a purely unsynchronized signal [3, Chapter 3.5]. This contributes to more predictable and stable behavior of the timer module.

- **Modular Design Approach:** The entire system was designed using a modular approach, breaking down the overall functionality into smaller, manageable, and independently verifiable Verilog modules (`clock_divider`, `debouncer`, `fsm`, `emergency_timer`, and `seven_seg_driver`). This strategy greatly simplified the design process, facilitated targeted debugging during simulation, and enhances code readability and maintainability. Furthermore, modularity promotes reusability of components in future projects.

- **Efficient 7-Segment Display Multiplexing:** To drive the 4-digit 7-segment display for the emergency timer, a multiplexing technique was implemented in the `seven_seg_driver` module. This involves rapidly activating one digit at a time (controlled by the 1 kHz clock, clk_{1kHz}) while sending the appropriate segment data for that digit. This approach significantly reduces the number of I/O pins required compared to dedicating separate segment lines for each of the four digits, and also simplifies the driving logic. The 1 kHz refresh rate is high enough to ensure a flicker-free display for the human eye.

- **Behavioral FSM Description with Defined State Encoding:** The Finite State Machine (`fsm`) was implemented using a behavioral Verilog description. While a specific binary encoding for the states was defined using `localparam` for clarity and debuggability (e.g., `NORMAL = 3'd4`, `EMERGENCY = 3'd0`), the transition and output logic were described using high-level constructs like `case` statements and `if-else` conditions. This approach still allows the FPGA synthesis tool considerable latitude to optimize the combinational logic implementing these functions based on the target Artix-7 architecture, aiming for efficient resource utilization and performance for the given state encoding.

These strategies collectively contribute to a more robust, efficient, and well-structured digital design.

4.3 Limitations and Future Improvements

While the implemented Automotive Tail Light System successfully meets the core design objectives and includes an extended emergency timer feature, certain limitations exist in the current design. These limitations also open avenues for potential future enhancements:

Current Limitations:

- **Absence of Reverse Obstacle Warning:** The current system activates a reverse light but does not include an audible warning (e.g., a buzzer) for nearby obstacles when reversing. This

functionality was intentionally omitted as it would typically require an ultrasonic sensor or similar proximity detection hardware, which was outside the scope and available components for this project.

- **Conditional Implementation of Position (Running) Lights:** The design intentionally does not include autonomously-activated position lights (also known as running lights or tail-lights) that operate at a lower intensity. In many modern vehicles, such lights are often automatically controlled based on ambient light conditions detected by a dedicated sensor. Lacking such a sensor for this project, a decision was made not to implement a simplified, manually-switched version of position lights to maintain focus on the core reactive signaling functionalities and the extended emergency timer feature. This leaves room for a more sophisticated, sensor-driven implementation in the future.
- **Fixed Blinking Rate:** The turn signals and hazard lights operate at a fixed 1 Hz blinking rate, as generated by the clock divider. There is no provision for varying this rate or for detecting a "bulb out" condition (which in real vehicles often results in a faster blinking rate).
- **Manual Emergency Mode Activation:** The emergency mode with the timer is activated by a specific button combination. In a real vehicle, advanced emergency stop signals (ESS) might be triggered automatically by sensors detecting very rapid deceleration or ABS activation.

Potential Future Improvements: Building upon the current design, several enhancements could be considered for future development:

- **Implementation of Sensor-Driven Position (Running) Lights:** A significant improvement would be to integrate an ambient light sensor. Based on its readings, the system could automatically activate position lights. This could be achieved by driving the main brake LEDs (left, right, and the central one) at a reduced intensity (e.g., using Pulse Width Modulation - PWM) when no other primary light function is active and low ambient light is detected. This would provide a more realistic and automated lighting feature.
- **Integration of Reverse Buzzer with Sensor Input:** If sensor hardware (e.g., an ultrasonic distance sensor interfaced with the FPGA) were available, the system could be extended to include an audible buzzer that activates when an obstacle is detected while in reverse. The frequency or pattern of the buzzer could vary with proximity.
- **Adaptive Blinking Rate / Bulb Out Detection:** The system could be enhanced to monitor the current drawn by the LED outputs (requiring additional circuitry). A significant change in current for a turn signal could indicate a "bulb out" (or LED failure) condition, and the FSM could then alter the blinking rate to a faster pace as an alert.
- **Dynamic Brake Light Modulation (Emergency Stop Signal - ESS):** For enhanced safety, upon very hard braking (which would require an accelerometer input or a specific "panic brake" signal), the brake lights could be programmed to flash rapidly for a short duration before becoming steady, a feature found in some modern ESS implementations.

- **Expansion with Front Lighting Control:** The modular design could be leveraged to expand the system to include control logic for front vehicle lights (headlights, daytime running lights, front turn signals), creating a more comprehensive vehicle lighting controller.
- **Enhanced Reset Synchronization for Timer:** While the current `emergency_timer` employs a single-flop synchronizer for its asynchronous reset, a future enhancement would be to implement a full two-flop synchronizer. This would provide a higher degree of protection against metastability, further increasing the reliability of the timer module, especially in environments with higher clock frequencies or more stringent reliability requirements.

These potential improvements could further increase the system's functionality, realism, and alignment with advanced automotive lighting features.

5 Conclusion

This project successfully demonstrated the design and implementation of an FPGA-based Automotive Tail Light System, incorporating core vehicle signaling functionalities along with an extended emergency timer feature. The development process encompassed theoretical design, Verilog HDL coding, simulation-based verification, and preparation for hardware deployment on a Xilinx Nexys FPGA platform.

5.1 Key Achievements and Lessons Learned

The successful completion of this project marks several key achievements. Primarily, a fully functional digital system was realized, adeptly managing essential automotive lights such as brake lights, independent turn signals, hazard lights, and a reverse light, with robust logic for handling concurrent operations. A significant extension to this core functionality was the integration of an innovative emergency mode, featuring an MM:SS timer displayed on a 7-segment interface, which showcased the ability to develop value-added features. This endeavor provided extensive, practical experience with Verilog HDL, from the detailed construction of individual modules like debouncers and clock dividers to the complex architectural design of a multi-state Finite State Machine. The project also reinforced a comprehensive understanding of the FPGA design workflow, including the critical stages of simulation using Icarus Verilog and GTKWave, the intricacies of the synthesis process, and the paramount importance of accurate constraint definition.

The development journey was also rich in lessons learned, largely through overcoming various design challenges. The process of refining the FSM logic through multiple iterations, developing effective testbench strategies, navigating subtle Verilog syntax issues, and addressing potential metastability in asynchronous signal paths—particularly through the implementation of a flip-flop synchronizer for the timer's reset—was instrumental in honing crucial problem-solving and debugging skills. The adopted modular design philosophy proved invaluable, not only in managing system complexity but also in facilitating targeted testing and significantly improving code maintainability and readability. Among the principal takeaways are the critical importance of conducting thorough simulation before committing to hardware, the meticulous attention to detail that HDL coding demands, a deepened appreciation for the practical implications of signal integrity issues such as debouncing and metastability, and the clear benefits derived from employing systematic design and verification methodologies. The inherently iterative nature of digital design, especially for intricate components like the FSM, also stands out as a significant practical insight.

5.2 Relevance to Real-World Applications

The developed Automotive Tail Light System, while a scaled prototype, holds direct and significant relevance to real-world automotive electronics. Modern vehicles rely heavily on sophisticated digital control systems for their lighting, which are fundamental for enhancing road safety and facilitating clear communication between road users. This project offers a practical insight into the type of complex logic embedded within automotive ECUs (Electronic Control Units), especially concerning the implementation of safety-critical signaling. Furthermore, features like the integrated emergency timer, though simplified in this context, allude to the broader concepts underpinning Advanced Driver-Assistance Systems (ADAS), where onboard technology provides enhanced situational information or initiates autonomous actions during critical events. The successful integration of multiple digital sub-systems—encompassing precise timing generation, state-based control logic, and display

driving mechanisms—mirrors a common and essential paradigm in the design of complex embedded systems prevalent throughout the automotive industry and in numerous other technological sectors.

Ultimately, the skills cultivated through this project, particularly in FPGA design, HDL programming, and digital system verification, are highly transferable and broadly applicable across diverse fields within electronics engineering. Beyond the immediate scope of automotive applications, these competencies are directly relevant to challenges in telecommunications, consumer electronics, and industrial automation, where the use of programmable logic is increasingly widespread. This project, therefore, serves as a robust practical foundation for engaging with more advanced digital design challenges in subsequent academic endeavors and future professional settings.

Acknowledgments

The author wishes to acknowledge the assistance provided by Google Gemini 2.5 Pro Prewiew and the DeepSeek AI model (via user interface) during the preparation of this project report. These AI tools were utilized for guidance on document structuring in LaTeX, help in solve some problems encountered in the development of the FSM and suggestions on report presentation

References

[1] T. L. Floyd, *Digital Fundamentals*, 11th. Pearson, 2015.

[2] N. Navet and F. Simonot-Lion, Eds., *Automotive Embedded Systems Handbook*. CRC Press, 2019, Chapter on Automotive Lighting.

[3] D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture*, RISC-V Edition. Morgan Kaufmann, 2021, Relevant sections on sequential logic, timing, and synchronization techniques.

[4] D. Inc., *Nexys 4 ddr fpga board reference manual*, 2020. [Online]. Available: `https://digilent.com/reference/programmable-logic/nexys-4-ddr/reference-manual` (visited on 10/27/2023).

A Appendix: FPGA Project Implementation Files

A.1 Verilog HDL Code

The complete Verilog HDL code for the Automotive Tail Light System is provided below. The project is structured into the following modules:

- `top.v`: The top-level module integrating all sub-components.
- `fsm.v`: The Finite State Machine controlling the light logic.
- `clock_divider.v`: Generates 1 Hz and 1 kHz clock signals.
- `debouncer.v`: Debounces button and switch inputs.
- `emergency_timer.v`: Implements the MM:SS emergency timer.
- `seven_seg_driver.v`: Drives the 7-segment display for the timer.

Listing 5: Complete Verilog code for the Automotive Tail Light System.

```
`timescale 1ns / 1ps

// Module to debounce a button input, ensuring a stable signal
module debouncer (
    input clk,
    input rst,
    input btn_in,
    output reg btn_out
);
    // Counter to measure input stability duration
    reg [20:0] counter;

    always @(posedge clk) begin
        if (rst) begin
            counter <= 0;
            btn_out <= 0;
        end
        // If input changes, start/continue counting.
        // Update output only after the input has been stable for a defined period.
        else if (btn_in != btn_out) begin
            counter <= counter + 1;
            if (counter == 21'd2_000_000)  btn_out <= btn_in; // Threshold for
                stability
        end
        // If input is stable or returns to previous state, reset counter.
        else counter <= 0;
    end
endmodule

// Generates slower clock signals (1Hz and 1kHz) from a faster input clock
module clock_divider (
    input clk,      // Input clock (100MHz)
    input rst,
    output reg clk_1Hz,
    output reg clk_1kHz
);
```

```
    // Counters to achieve desired frequencies by toggling output at specific counts
    reg [26:0] counter_1Hz;         // For 50_000_000 counts (100MHz / 2*1Hz)
    reg [16:0] counter_1kHz;        // For 50_000 counts (100MHz / 2*1kHz)

    // 1Hz generator
    always @(posedge clk) begin
        if (rst) begin
            counter_1Hz <= 0;
            clk_1Hz <= 0;
        end
        // Toggle clk_1Hz and reset counter when half period is reached
        else if (counter_1Hz == 27'd50_000_000 -1) begin
            clk_1Hz <= ~clk_1Hz;
            counter_1Hz <= 0;
        end
        else
            counter_1Hz <= counter_1Hz + 1;
    end

    // 1kHz generator
    always @(posedge clk) begin
        if(rst) begin
            counter_1kHz <= 0;
            clk_1kHz <= 0;
        end
        // Toggle clk_1kHz and reset counter when half period is reached
        else if (counter_1kHz == 17'd50_000 -1) begin
            clk_1kHz <= ~clk_1kHz;
            counter_1kHz <= 0;
        end
        else
            counter_1kHz <= counter_1kHz + 1;
    end
endmodule

// Finite State Machine to control vehicle light logic
module fsm (
    input clk,
    input rst,
    input brake_btn,
    input emergency_btn,
    input reverse_sw,
    input left_sw,
    input right_sw,
    input four_way_sw,
    input blink,                    // 1Hz clock for blinking
    // LED outputs: [reverse, brake_L, brake_R, turn_L, turn_R]
    output reg [4:0] leds,
    output reg emergency_active
);

    // State definitions
    localparam[2:0]
     EMERGENCY  = 3'd0,
     LEFT_TURN  = 3'd1,
     RIGHT_TURN = 3'd2,
     FOUR_WAY   = 3'd3,
     NORMAL     = 3'd4;
```

```
94
95      reg [2:0] current_state, next_state;
96
97      // Internal signal for turn indicator LEDs {left, right}
98      reg [1:0] turn_leds;
99
100     // Sequential logic for state register
101     always @(posedge clk) begin
102         if (rst) current_state <= NORMAL;
103         else current_state <= next_state;
104     end
105
106     // Combinational logic for state transitions
107     always @(*) begin
108         next_state = current_state; // Default: remain in current state
109
110         // Emergency button toggles EMERGENCY state and has highest priority
111         if (emergency_btn) begin
112             next_state = (current_state == EMERGENCY) ? NORMAL : EMERGENCY;
113         end
114         // Logic for other states if not in or transitioning to/from EMERGENCY
115         else if (current_state != EMERGENCY) begin
116             case (current_state)
117                 NORMAL: begin // From NORMAL, can go to turn signals or four-way
118                     if (four_way_sw)       next_state = FOUR_WAY;
119                     else if (left_sw)      next_state = LEFT_TURN;
120                     else if (right_sw)     next_state = RIGHT_TURN;
121                 end
122                 // Return to NORMAL if respective switch is turned off
123                 LEFT_TURN:  if (!left_sw && !four_way_sw)   next_state = NORMAL;
124                 RIGHT_TURN: if (!right_sw && !four_way_sw)  next_state = NORMAL;
125                 FOUR_WAY:   if (!four_way_sw)               next_state = NORMAL;
126             endcase
127         end
128     end
129
130     // Combinational logic for FSM outputs (LEDs and emergency status)
131     always @(*) begin
132         leds = 5'b00000; // Default all off
133         emergency_active = 0;
134         turn_leds = 2'b00;
135
136         // Determine turn signal behavior based on state
137         case (current_state)
138             LEFT_TURN:  turn_leds = {blink, 1'b0}; // Left blinks
139             RIGHT_TURN: turn_leds = {1'b0, blink}; // Right blinks
140             FOUR_WAY:   turn_leds = {blink, blink}; // Both blink
141             EMERGENCY:  turn_leds = {blink, blink}; // Both blink
142             default:    turn_leds = 2'b00;
143         endcase
144
145         // Output logic for LEDs based on state and inputs
146         if (current_state == EMERGENCY) begin
147             emergency_active = 1;
148             // In EMERGENCY: turn signals blink, brake lights are solid ON
149             leds[1] = turn_leds[1]; // turn_L
150             leds[0] = turn_leds[0]; // turn_R
151             leds[3] = 1'b1;         // brake_L
```

```
152             leds[2] = 1'b1;          // brake_R
153             leds[4] = reverse_sw;    // reverse light
154         end else begin // Normal operation
155             emergency_active = 0;
156             leds[1] = turn_leds[1]; // turn_L
157             leds[0] = turn_leds[0]; // turn_R
158             if (brake_btn) begin     // Brake lights active if brake_btn pressed
159                 leds[3] = 1'b1;
160                 leds[2] = 1'b1;
161             end
162             leds[4] = reverse_sw;    // reverse light
163         end
164     end
165 endmodule
166 
167 // Timer to count minutes and seconds, active during emergency mode
168 module emergency_timer (
169     input clk,               // Clocked by 1Hz for second counting
170     input reset_as,          // Asynchronous reset (active high to reset timer)
171     input enable,            // Enables counting when emergency is active
172     output reg [3:0] minutes,
173     output reg [5:0] seconds
174 );
175     // Synchronize asynchronous reset to the local clock domain
176     reg reset_sync;
177     always @(posedge clk) reset_sync <= reset_as;
178 
179     // Counter logic for seconds and minutes (up to 15:59)
180     always @(posedge clk) begin
181         if (reset_sync || !enable) begin // Reset if reset asserted or not enabled
182             minutes <= 0;
183             seconds <= 0;
184         end
185         else if (enable) begin // Count when enabled
186             if (seconds == 59) begin
187                 seconds <= 0;
188                 minutes <= (minutes == 15) ? 0 : minutes + 1; // Minutes roll over
                        at 15
189             end
190             else seconds <= seconds + 1;
191         end
192     end
193 endmodule
194 
195 // Drives a 4-digit 7-segment display to show minutes and seconds
196 module seven_seg_driver (
197     input clk_1kHz,          // Clock for display refresh and multiplexing
198     input rst,
199     input enable,            // Enables the display output
200     input [3:0] minutes,
201     input [5:0] seconds,
202     output reg [7:0] AN,     // Anode control (active low)
203     output reg [6:0] segments // Segment control (active low)
204 );
205     reg [1:0] digit_sel;     // Selects one of the 4 digits to activate
206     reg [3:0] digit_value;   // BCD value for the selected digit
207 
208     // Internal registers for synchronizing time inputs
```

```
    reg [3:0] minutes_reg;
    reg [5:0] seconds_reg;

    // Convert registered time to BCD for each digit
    wire [3:0] min_tens = minutes_reg / 10;
    wire [3:0] min_units = minutes_reg % 10;
    wire [3:0] sec_tens = seconds_reg / 10;
    wire [3:0] sec_units = seconds_reg % 10;

    // Main logic for display multiplexing and BCD-to-7-segment conversion
    always @(posedge clk_1kHz) begin
        if (rst || !enable) begin // If reset or disabled, turn off display
            digit_sel <= 2'd0;
            AN <= 8'b11111111;      // All anodes off
            segments <= 7'b1111111; // All segments off
            minutes_reg <= 0;
            seconds_reg <= 0;
        end
        else begin
            minutes_reg <= minutes; // Latch current time
            seconds_reg <= seconds;

            AN <= 8'b11111111; // Briefly turn off anodes to prevent ghosting

            // Select active digit and its BCD value based on digit_sel
            case (digit_sel)
                2'd0: begin AN <= 8'b11111110; digit_value = sec_units; end
                2'd1: begin AN <= 8'b11111101; digit_value = sec_tens;  end
                2'd2: begin AN <= 8'b11111011; digit_value = min_units; end
                2'd3: begin AN <= 8'b11110111; digit_value = min_tens;  end
            endcase

            // BCD to 7-segment decoder (common anode, active low segments)
            case (digit_value)
                4'h0: segments <= 7'b1000000; // 0
                4'h1: segments <= 7'b1111001; // 1
                4'h2: segments <= 7'b0100100; // 2
                4'h3: segments <= 7'b0110000; // 3
                4'h4: segments <= 7'b0011001; // 4
                4'h5: segments <= 7'b0010010; // 5
                4'h6: segments <= 7'b0000010; // 6
                4'h7: segments <= 7'b1111000; // 7
                4'h8: segments <= 7'b0000000; // 8
                4'h9: segments <= 7'b0010000; // 9
                default: segments <= 7'b1111111; // Blank
            endcase

            digit_sel <= digit_sel + 1; // Cycle to the next digit
        end
    end
endmodule

// Top-level module: integrates debouncers, clock divider, FSM, timer, and display
    driver
module top (
    input CLK100MHZ,
    input BTNC, BTNU, BTNL, // Buttons: Center (brake), Up, Left
    input [15:0] swt,
```

```
    input rst,
    output [15:0] led,
    output [7:0] AN,
    output [6:0] segments
);
    // Internal signals connecting the modules
    wire clk_1Hz, clk_1kHz;
    wire brake_debounced, btnu_debounced, btnl_debounced, reverse_debounced;
    wire [4:0] leds_fsm;    // LED control signals from FSM
    wire [3:0] minutes;
    wire [5:0] seconds;
    wire emergency_active;

    // Instantiate clock divider for 1Hz (blink) and 1kHz (display refresh)
    clock_divider clk_div(CLK100MHZ, rst, clk_1Hz, clk_1kHz);

    // Instantiate debouncers for critical inputs
    debouncer deb_brake(CLK100MHZ, rst, BTNC, brake_debounced);
    debouncer deb_reverse(CLK100MHZ, rst, swt[8], reverse_debounced); // swt[8] as
        reverse
    debouncer deb_u(CLK100MHZ, rst, BTNU, btnu_debounced);
    debouncer deb_l(CLK100MHZ, rst, BTNL, btnl_debounced);
    // Emergency is triggered by BTNU AND BTNL
    wire emergency_btn = btnu_debounced & btnl_debounced;

    // Instantiate the main FSM for light control logic
    fsm fsm_inst(
        .clk(CLK100MHZ), // FSM clocked by main system clock
        .rst(rst),
        .brake_btn(brake_debounced),
        .emergency_btn(emergency_btn),
        .reverse_sw(reverse_debounced),
        .left_sw(swt[15]),
        .right_sw(swt[0]),
        .four_way_sw(swt[7]),
        .blink(clk_1Hz),
        .leds(leds_fsm),            // FSM's internal LED representation
        .emergency_active(emergency_active)
    );

    // Instantiate emergency timer: runs on 1Hz, enabled by emergency_active, and
        reset when emergency is not active.
    emergency_timer timer(clk_1Hz, ~emergency_active, emergency_active, minutes,
        seconds);

    // Instantiate 7-segment display driver: refreshed by 1kHz, displays timer
        values, enabled by emergency_active.
    seven_seg_driver display(clk_1kHz, rst, emergency_active, minutes, seconds, AN,
        segments);

    // Map FSM's logical LED outputs to physical board LEDs
    // FSM leds_fsm mapping: [4:reverse, 3:brake_L, 2:brake_R, 1:turn_L, 0:turn_R]
    assign led[0]  = leds_fsm[2];               // Physical led[0] = Right Brake
    assign led[15] = leds_fsm[3];               // Physical led[15] = Left Brake
    assign led[7]  = leds_fsm[2] | leds_fsm[3]; // Physical led[7] = Center Brake
    assign led[1]  = leds_fsm[0];               // Physical led[1] = Right Turn
    assign led[14] = leds_fsm[1];               // Physical led[14] = Left Turn
    assign led[6]  = leds_fsm[4];               // Physical led[6] = Reverse
```

```
endmodule
```

A.2 XDC Constraints File

The Xilinx Design Constraints (XDC) file used for mapping the top-level signals to the physical pins of the Nexys 4 DDR FPGA board and for defining clock properties is shown below.

Listing 6: XDC Constraints file for the project.

```
## Clock
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports CLK100MHZ];
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
    CLK100MHZ];

## Buttuns
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports BTNC];
    #Brake buttun
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports rst];
    #Reset buttun
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports BTNU];
    #Emergency buttun
set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports BTNL];
    #Emergency buttun

## Switch
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports {swt[0]}];
    #Right turn switch
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports {swt[1]}];
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports {swt[2]}];
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports {swt[3]}];
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports {swt[4]}];
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports {swt[5]}];
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports {swt[6]}];
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports {swt[7]}];
    #Four wat switch
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS33 } [get_ports {swt[8]}];
    #Reverse switch
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS33 } [get_ports {swt[9]}];
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports {swt[10]}];
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports {swt[11]}];
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports {swt[12]}];
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports {swt[13]}];
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports {swt[14]}];
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports {swt[15]}];
    #Left turn switch

## LED
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports {led[0]}];
    #Right brake
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports {led[1]}];
    #Right turn
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports {led[2]}];
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports {led[3]}];
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports {led[4]}];
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {led[5]}];
```

```
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports {led[6]}];
    # Reverse
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {led[7]}];
    # Central brake
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {led[8]}];
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports {led[9]}];
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {led[10]}];
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports {led[11]}];
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {led[12]}];
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {led[13]}];
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports {led[14]}];
    #Left turn
set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports {led[15]}];
    #Left brake

## 7-segment display
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports {segments
    [0]}];
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports {segments
    [1]}];
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports {segments
    [2]}];
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports {segments
    [3]}];
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports {segments
    [4]}];
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports {segments
    [5]}];
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports {segments
    [6]}];
set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports {AN[0]}];
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports {AN[1]}];
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports {AN[2]}];
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports {AN[3]}];
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports {AN[4]}];
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports {AN[5]}];
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports {AN[6]}];
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports {AN[7]}];
```

B Appendix: Testbench Code and Simulation Approach

This appendix provides the Verilog HDL code for the primary testbench used to simulate and verify the functionality of the Automotive Tail Light System. The testbench was designed to instantiate the top-level module (`top`) of the automotive tail light system and apply a sequence of stimuli to its inputs to cover various operational scenarios. The simulation output was typically observed using GTKWave by dumping signals to a VCD (Value Change Dump) file. The testbench also included `$display` statements for critical checkpoints (though full logs are not reproduced here).

Listing 7: Verilog testbench for the Automotive Tail Light System (`testbench_automotive_light.v`).

```
`timescale 1ns / 1ps

// Testbench module
// Prints "PASSED" or "FAILED" messages for each test.

module tb();

    // --------------------
    // 1) Clock Generation
    // --------------------
    // Main FSM clock (10ns period -> 100MHz simulated)
    reg clk_main = 0;
    always #5 clk_main = ~clk_main;

    // "Blink" clock (simulates 1Hz, accelerated: 20ns period)
    reg blink = 0;
    always #10 blink = ~blink;

    // Display clock (simulates 1kHz, accelerated: 2ns period)
    reg clk_disp = 0;
    always #1 clk_disp = ~clk_disp;

    // --------------------
    // 2) Reset and Input Signals
    // --------------------
    reg rst_fsm  = 1;   // FSM reset
    reg rst_disp = 1;   // Display driver reset

    reg brake_btn      = 0;
    reg reverse_sw     = 0;
    reg left_sw        = 0;
    reg right_sw       = 0;
    reg four_way_sw    = 0;
    reg btnu_debounced = 0;
    reg btnl_debounced = 0;

    // Emergency button is the AND of btnu_debounced and btnl_debounced
    wire emergency_btn = btnu_debounced & btnl_debounced;

    // --------------------
    // 3) FSM Outputs
    // --------------------
    // leds_fsm: [reverse, brake_left, brake_right, turn_left, turn_right]
    wire [4:0] leds_fsm;
    wire       emergency_active;
```

```
46
47      // --------------------
48      // 4) Emergency Timer Outputs
49      // --------------------
50      wire [3:0] minutes;
51      wire [5:0] seconds;
52
53      // --------------------
54      // 5) Display Driver Outputs
55      // --------------------
56      wire [7:0] AN;
57      wire [6:0] segments;
58
59      // --------------------
60      // 6) GTKWave Dump (optional)
61      // --------------------
62      initial begin
63          $dumpfile("waveform_tb.vcd");
64          $dumpvars(0, tb);
65      end
66
67      // --------------------
68      // 7) Instantiate FSM
69      // --------------------
70      fsm dut_fsm (
71          .clk(clk_main),
72          .rst(rst_fsm),
73          .brake_btn(brake_btn),
74          .emergency_btn(emergency_btn),
75          .reverse_sw(reverse_sw),
76          .left_sw(left_sw),
77          .right_sw(right_sw),
78          .four_way_sw(four_way_sw),
79          .blink(blink),
80          .leds(leds_fsm),
81          .emergency_active(emergency_active)
82      );
83
84      // --------------------
85      // 8) Instantiate Emergency Timer
86      //    - blink as clock (1Hz simulated)
87      //    - asynchronous reset = !emergency_active
88      // --------------------
89      emergency_timer dut_timer (
90          .clk(blink),
91          .reset_as(~emergency_active), // Active low reset when emergency is NOT
                active
92          .enable(emergency_active),
93          .minutes(minutes),
94          .seconds(seconds)
95      );
96
97      // --------------------
98      // 9) Instantiate Display Driver
99      //    - clk_disp as 1kHz simulated clock
100     // --------------------
101     seven_seg_driver dut_display (
102         .clk_1kHz(clk_disp),
```

```
        .rst(rst_disp),
        .enable(emergency_active),
        .minutes(minutes),
        .seconds(seconds),
        .AN(AN),
        .segments(segments)
    );

    // --------------------
    // 10) FSM Tests
    // --------------------
    initial begin
        $display("\n============== START FSM TESTS ==============\n");

        // 10.1) Initial RESET
        #0    rst_fsm = 1;
              brake_btn     = 0;
              reverse_sw    = 0;
              left_sw       = 0;
              right_sw      = 0;
              four_way_sw   = 0;
              btnu_debounced = 0;
              btnl_debounced = 0;
        #30   rst_fsm = 0;
        #1    $display("[1) RESET] Inputs = 0, Expected leds_fsm = 00000, emergency_
            active = 0");
        #1    if (leds_fsm == 5'b00000 && emergency_active == 0)
                 $display("    --> TEST RESET PASSED\n");
              else
                 $display("    --> TEST RESET FAILED: leds_fsm = %05b, emergency_
                     active = %b\n", leds_fsm, emergency_active);

        // 10.2) Test BRAKE (brake_btn)
        #20   brake_btn = 1;
        #5    $display("[2) BRAKE ON] brake_btn=1, Expected brake_right=1, brake_
            left=1");
        #1    if (leds_fsm[2] == 1 && leds_fsm[3] == 1) // leds_fsm[3]=brake_left,
            leds_fsm[2]=brake_right
                 $display("    --> TEST BRAKE PASSED\n");
              else
                 $display("    --> TEST BRAKE FAILED: leds_fsm = %05b\n", leds_fsm);
        #10   brake_btn = 0;
        #1    $display("[2) BRAKE OFF] brake_btn=0, Expected leds_fsm = 00000");
        #1    if (leds_fsm == 5'b00000)
                 $display("    --> TEST BRAKE OFF PASSED\n");
              else
                 $display("    --> TEST BRAKE OFF FAILED: leds_fsm = %05b\n", leds_
                     fsm);

        // 10.3) Test LEFT TURN SIGNAL (LEFT_TURN)
        #20   left_sw = 1;
        #5    $display("[3) LEFT TURN ON] left_sw=1, Expected turn_left blinking");
        // Check over 3 blink cycles
        repeat (3) begin
            @(posedge blink);
            #1 $display("    blink=%b -> turn_left=%b, leds_fsm = %05b",
                        blink, leds_fsm[1], leds_fsm); // leds_fsm[1]=turn_left
        end
```

```
156         // Verify blinking by checking current state (0 or 1 is acceptable for
                simplicity)
157         #1    if (leds_fsm[1] == 0 || leds_fsm[1] == 1)
158                  $display("    --> TEST LEFT TURN PASSED\n");
159               else
160                  $display("    --> TEST LEFT TURN FAILED: leds_fsm = %05b\n", leds_
                         fsm);
161         #10   left_sw = 0;
162         #1    $display("[3) LEFT TURN OFF] left_sw=0, Expected leds_fsm = 00000");
163         #1    if (leds_fsm == 5'b00000)
164                  $display("    --> TEST LEFT TURN OFF PASSED\n");
165               else
166                  $display("    --> TEST LEFT TURN OFF FAILED: leds_fsm = %05b\n",
                         leds_fsm);
167
168         // 10.4) Test RIGHT TURN SIGNAL (RIGHT_TURN)
169         #20   right_sw = 1;
170         #5    $display("[4) RIGHT TURN ON] right_sw=1, Expected turn_right blinking"
                );
171         repeat (3) begin
172             @(posedge blink);
173             #1 $display("    blink=%b -> turn_right=%b, leds_fsm = %05b",
174                         blink, leds_fsm[0], leds_fsm); // leds_fsm[0]=turn_right
175         end
176         #1    if (leds_fsm[0] == 0 || leds_fsm[0] == 1)
177                  $display("    --> TEST RIGHT TURN PASSED\n");
178               else
179                  $display("    --> TEST RIGHT TURN FAILED: leds_fsm = %05b\n", leds_
                         fsm);
180         #10   right_sw = 0;
181         #1    $display("[4) RIGHT TURN OFF] right_sw=0, Expected leds_fsm = 00000");
182         #1    if (leds_fsm == 5'b00000)
183                  $display("    --> TEST RIGHT TURN OFF PASSED\n");
184               else
185                  $display("    --> TEST RIGHT TURN OFF FAILED: leds_fsm = %05b\n",
                         leds_fsm);
186
187         // 10.5) Test HAZARD (FOUR_WAY)
188         #20   four_way_sw = 1;
189         #5    $display("[5) HAZARD ON] four_way_sw=1, Expected turn_left+turn_right
                blinking simultaneously");
190         repeat (3) begin
191             @(posedge blink);
192             #1 $display("    blink=%b -> turn_left=%b, turn_right=%b, leds_fsm = %05
                    b",
193                         blink, leds_fsm[1], leds_fsm[0], leds_fsm);
194         end
195         #1    if ((leds_fsm[1] == 0 || leds_fsm[1] == 1) &&
196                   (leds_fsm[0] == 0 || leds_fsm[0] == 1))
197                  $display("    --> TEST HAZARD PASSED\n");
198               else
199                  $display("    --> TEST HAZARD FAILED: leds_fsm = %05b\n", leds_fsm)
                         ;
200         #10   four_way_sw = 0;
201         #1    $display("[5) HAZARD OFF] four_way_sw=0, Expected leds_fsm = 00000");
202         #1    if (leds_fsm == 5'b00000)
203                  $display("    --> TEST HAZARD OFF PASSED\n");
204               else
```

```
205                 $display("    --> TEST HAZARD OFF FAILED: leds_fsm = %05b\n", leds_
                        fsm);
206
207         // 10.6) Test REVERSE
208         #20   reverse_sw = 1;
209         #5    $display("[6) REVERSE ON] reverse_sw=1, Expected reverse = 1 (leds_fsm
                [4])");
210         #1    if (leds_fsm[4] == 1)
211                 $display("    --> TEST REVERSE PASSED\n");
212               else
213                 $display("    --> TEST REVERSE FAILED: leds_fsm = %05b\n", leds_fsm
                        );
214         #10   reverse_sw = 0;
215         #1    $display("[6) REVERSE OFF] reverse_sw=0, Expected leds_fsm = 00000");
216         #1    if (leds_fsm == 5'b00000)
217                 $display("    --> TEST REVERSE OFF PASSED\n");
218               else
219                 $display("    --> TEST REVERSE OFF FAILED: leds_fsm = %05b\n", leds
                        _fsm);
220
221         // 10.7) Test CONCURRENT (brake + left turn)
222         #20   brake_btn = 1; left_sw = 1;
223         #5    $display("[7) CONCURRENT ON] brake=1 + left=1, Expected: turn_left
                blinking + brakes steady");
224         repeat (3) begin
225             @(posedge blink);
226             #1 $display("    blink=%b -> turn_left=%b, brake_right=%b, brake_left=%b
                    , leds_fsm = %05b",
227                        blink, leds_fsm[1], leds_fsm[2], leds_fsm[3], leds_fsm);
228         end
229         // Verify brakes are on (1) and turn_left is blinking (0 or 1)
230         #1    if ((leds_fsm[2] == 1 && leds_fsm[3] == 1) && (leds_fsm[1] == 0 ||
                leds_fsm[1] == 1))
231                 $display("    --> TEST CONCURRENT PASSED\n");
232               else
233                 $display("    --> TEST CONCURRENT FAILED: leds_fsm = %05b\n", leds_
                        fsm);
234         #10   brake_btn = 0; left_sw = 0;
235         #1    $display("[7) CONCURRENT OFF] brake=0 + left=0, Expected leds_fsm =
                00000");
236         #1    if (leds_fsm == 5'b00000)
237                 $display("    --> TEST CONCURRENT OFF PASSED\n");
238               else
239                 $display("    --> TEST CONCURRENT OFF FAILED: leds_fsm = %05b\n",
                        leds_fsm);
240
241         // 10.8) Test EMERGENCY Mode
242         $display("\n[8) Test EMERGENCY Mode]\n");
243         #20   btnu_debounced = 1; btnl_debounced = 1;
244         #5    $display("[8.1) EMERGENCY ON] btnu=1 + btnl=1, Expected: emergency_
                active = 1");
245         #1    if (emergency_active == 1)
246                 $display("    --> TEST EMERGENCY ON PASSED\n");
247               else
248                 $display("    --> TEST EMERGENCY ON FAILED: emergency_active = %b\n
                        ", emergency_active);
249
250         // During emergency, also activate reverse
```

```
251         #20   reverse_sw = 1;
252         #5    $display("[8.2) EMER + REVERSE] reverse=1, Expected: reverse=1 + turns
                 blinking + brakes steady");
253         repeat (3) begin
254             @(posedge blink);
255             #1 $display("    blink=%b -> turn_left=%b, turn_right=%b, brake_right=%b
                    , brake_left=%b, reverse=%b, leds_fsm = %05b",
256                         blink, leds_fsm[1], leds_fsm[0], leds_fsm[2], leds_fsm[3],
                                leds_fsm[4], leds_fsm);
257         end
258         // Verify "brakes steady" and "reverse on" and "turns blinking"
259         #1    if ((leds_fsm[2] == 1 && leds_fsm[3] == 1 && leds_fsm[4] == 1) &&
260                   ((leds_fsm[1] == 0) || (leds_fsm[1] == 1)) &&
261                   ((leds_fsm[0] == 0) || (leds_fsm[0] == 1)))
262               $display("    --> TEST EMER + REVERSE PASSED\n");
263             else
264               $display("    --> TEST EMER + REVERSE FAILED: leds_fsm = %05b\n",
                      leds_fsm);
265
266         // Deactivate emergency
267         #20   btnu_debounced = 0; btnl_debounced = 0; reverse_sw = 0;
268         #5    $display("[8.3) EMERGENCY OFF] btnu=0 + btnl=0, Expected: return to
                NORMAL state, leds_fsm = 00000");
269         #1    if (emergency_active == 0 && leds_fsm == 5'b00000)
270               $display("    --> TEST EMERGENCY OFF PASSED\n");
271             else
272               $display("    --> TEST EMERGENCY OFF FAILED: emergency_active = %b,
                     leds_fsm = %05b\n", emergency_active, leds_fsm);
273
274         $display("\n============== END FSM TESTS ==============\n");
275     end
276
277     // --------------------
278     // 11) Emergency Timer Tests
279     // --------------------
280     initial begin
281         // Wait for FSM tests to progress so emergency mode can be entered/exited
282         #200; // Adjust delay if FSM tests take longer
283         $display("\n=========== START EMERGENCY_TIMER TESTS ==========\n");
284
285         // 11.1) Initially emergency_active = 0, timer should remain at zero
286         #1    $display("[11.1) TIMER RESET] Expected: minutes=0, seconds=0");
287         #1    if (minutes == 0 && seconds == 0)
288               $display("    --> TEST TIMER RESET PASSED\n");
289             else
290               $display("    --> TEST TIMER RESET FAILED: minutes=%0d, seconds=%0d
                      \n", minutes, seconds);
291
292         // 11.2) Force emergency_active=1 to start counting
293         force dut_fsm.emergency_active = 1;
294         #5    $display("[11.2) TIMER START] emergency_active=1, Timer should start
                counting");
295
296         // Wait 70 blink cycles -> seconds = 70 mod 60 = 10, minutes = 1
297         repeat (70) @(posedge blink);
298         #1    $display("[11.2) After 70 cycles] Expected: seconds=10, minutes=1");
299         #1    if (seconds == 10 && minutes == 1)
300               $display("    --> TEST TIMER 70s PASSED\n");
```

```
301                 else
302                     $display("    --> TEST TIMER 70s FAILED: seconds=%0d, minutes=%0d\n
                        ", seconds, minutes);
303
304         // Wait another 50 cycles -> total 120 -> seconds=0, minutes=2
305         repeat (50) @(posedge blink);
306         #1    $display("[11.2) After 120 cycles] Expected: seconds=0, minutes=2");
307         #1    if (seconds == 0 && minutes == 2)
308                     $display("    --> TEST TIMER 120s PASSED\n");
309               else
310                     $display("    --> TEST TIMER 120s FAILED: seconds=%0d, minutes=%0d\
                        n", seconds, minutes);
311
312         // 11.3) Disable emergency -> timer resets
313         force dut_fsm.emergency_active = 0;
314         #5    $display("[11.3) TIMER RESET (EMERGENCY OFF)] Expected: seconds=0,
                  minutes=0");
315         #1    if (seconds == 0 && minutes == 0)
316                     $display("    --> TEST TIMER RESET (EMERGENCY OFF) PASSED\n");
317               else
318                     $display("    --> TEST TIMER RESET (EMERGENCY OFF) FAILED: seconds
                        =%0d, minutes=%0d\n", seconds, minutes);
319         release dut_fsm.emergency_active;
320
321         $display("\n=========== END EMERGENCY_TIMER TESTS ===========\n");
322     end
323
324     // --------------------
325     // 12) Display Driver (seven_seg_driver) Tests
326     // --------------------
327     initial begin
328         // Wait for timer to have produced minutes and seconds values
329         #400; // Adjust delay if previous tests take longer
330         $display("\n======== START SEVEN_SEG_DRIVER TESTS ========\n");
331
332         // Enable display and remove its reset
333         rst_disp = 0;                       // Deactivate display reset
334         force dut_fsm.emergency_active = 1; // Force display enable (via emergency_
                active)
335
336         // 12.1) Verify 4-digit multiplexing by synchronizing with clk_disp
337         // Wait for 4 posedges of clk_disp to sample anodes stably
338         @(posedge clk_disp); // Activates AN0 (digit_sel=0)
339         #0.1; // Small delay for stabilization
340         $display("[12.1) DIGIT 0] Expected: AN = 11111110 (only AN0 active)");
341         if (AN === 8'b11111110)
342             $display("    --> TEST DIGIT 0 PASSED\n");
343         else
344             $display("    --> TEST DIGIT 0 FAILED: AN = %08b\n", AN);
345
346         @(posedge clk_disp); // Activates AN1 (digit_sel=1)
347         #0.1;
348         $display("[12.1) DIGIT 1] Expected: AN = 11111101 (only AN1 active)");
349         if (AN === 8'b11111101)
350             $display("    --> TEST DIGIT 1 PASSED\n");
351         else
352             $display("    --> TEST DIGIT 1 FAILED: AN = %08b\n", AN);
353
```

```
354         @(posedge clk_disp); // Activates AN2 (digit_sel=2)
355         #0.1;
356         $display("[12.1) DIGIT 2] Expected: AN = 11111011 (only AN2 active)");
357         if (AN === 8'b11111011)
358             $display("    --> TEST DIGIT 2 PASSED\n");
359         else
360             $display("    --> TEST DIGIT 2 FAILED: AN = %08b\n", AN);
361 
362         @(posedge clk_disp); // Activates AN3 (digit_sel=3)
363         #0.1;
364         $display("[12.1) DIGIT 3] Expected: AN = 11110111 (only AN3 active)");
365         if (AN === 8'b11110111)
366             $display("    --> TEST DIGIT 3 PASSED\n");
367         else
368             $display("    --> TEST DIGIT 3 FAILED: AN = %08b\n", AN);
369 
370         // 12.2) Reset display and verify it's off
371         release dut_fsm.emergency_active; // Release force
372         rst_disp = 1;                     // Apply reset
373         @(posedge clk_disp);              // Wait for signal update
374         #0.1;
375         $display("[12.2) DISPLAY RESET] Expected: AN = 11111111, segments = 1111111
                (all off)");
376         if (AN === 8'b11111111 && segments === 7'b1111111) // Assuming active-low
                segments turn off to 1s
377             $display("    --> TEST DISPLAY RESET PASSED\n");
378         else
379             $display("    --> TEST DISPLAY RESET FAILED: AN = %08b, segments = %07b\
                    n", AN, segments);
380 
381         $display("\n======== END SEVEN_SEG_DRIVER TESTS ========\n");
382         #20 $finish;
383     end
384 
385 endmodule
```

C Appendix: FPGA Platform and Utilized On-Board Peripherals

This project was implemented on the Xilinx Nexys 4 DDR (Artix-7 FPGA) development board. The design made use of several on-board peripherals, the specifications and detailed operational characteristics of which are documented in the board's official reference manual, cited as [4] in the References section.

The key on-board components utilized in this project include:

- **System Clock (CLK100MHZ):**
  - **Description:** A 100 MHz crystal oscillator providing the primary clock signal for the FPGA design.
  - **Usage:** Served as the master clock from which all other required clock frequencies (1 Hz, 1 kHz) were derived using the `clock_divider` module.
- **Push Buttons (BTNC, BTNU, BTNL, and CPU Reset Button for 'rst'):**
  - **Description:** Momentary contact push buttons.
  - **Usage:** BTNC was used for the brake signal. BTNU and BTNL were used in combination for activating the emergency mode. The CPU Reset button was used as the global asynchronous reset for the system. All button inputs were processed by `debouncer` modules.
- **Slide Switches (swt[15], swt[8], swt[7], swt[0]):**
  - **Description:** Multi-position slide switches.
  - **Usage:** Used to simulate driver inputs: `swt[15]` for the left turn signal, `swt[0]` for the right turn signal, `swt[7]` for the four-way hazard lights, and `swt[8]` for the reverse gear engagement (this input was also debounced).
- **Discrete LEDs (led[15:0]):**
  - **Description:** Individual Light Emitting Diodes.
  - **Usage:** Specific LEDs were used to indicate the status of the tail light functions: left-/right brake lights (`led[15]`, `led[0]`), left/right turn signals (`led[14]`, `led[1]`), reverse light (`led[6]`), and an additional combined brake indicator (`led[7]`).
- **Four-Digit Seven-Segment Display (AN[7:0], segments[6:0]):**
  - **Description:** Common anode, multiplexed 7-segment display.
  - **Usage:** Used to display the elapsed time (MM:SS) for the emergency timer feature, driven by the `seven_seg_driver` module.

For precise pin assignments of these peripherals to the FPGA, refer to the XDC Constraints file provided in Appendix A.2. Detailed electrical characteristics and further usage notes can be found in the aforementioned board reference manual [4].