EE/CSE 371
Dec 6, 2025
Lab 6 Report

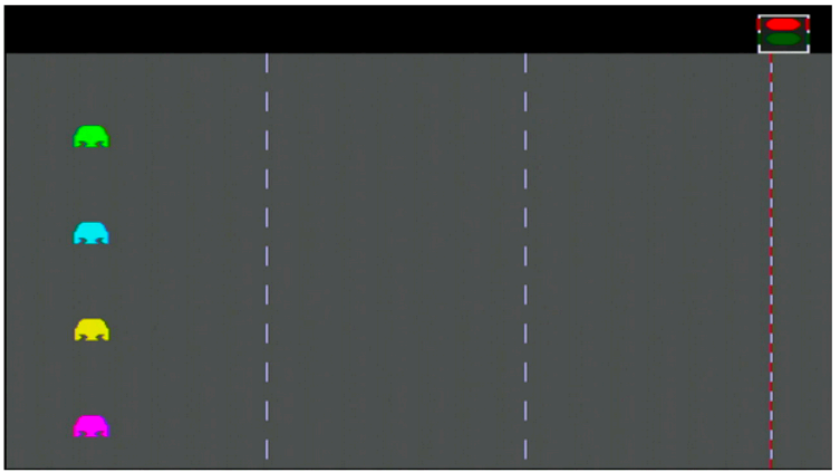
Lab 6

Video demo of project: 371_final_project_demo.mp4

Design Procedure

This lab implements the game Red Light Green Light, an interactive game with synchronized control and display logic, in hardware on an FPGA using a modular FSM and datapath design to understand how hardware executes real time algorithms with inputs, randomness and a visual VGA output. A player controls a car using board switches to move forward, up, or down while 3 AI opponents move automatically using a random sequences generated by an LFSR. A traffic light system switches between red and green at random intervals signaling when the player can safely move towards the finish line. The game controller manages all phases of the game: start screen, idle waiting, resetting positions, gameplay, round end display, inter-round transitions and score display. The player is disqualified from a round for moving during red light. The scores per round are tracked for player and the other AI cars. The position updates and movement detection are handled by their own respective modules. A VGA draw controller, provided by the teaching team, renders all sprites: cars, traffic lights, scores, and effects such as flashing any disqualified players. Scores and rounds are displayed on the HEX displays and LEDs. The design utilizes memory, input handling, random behavior, multi object tracking, multiple FSM control, and VGA graphic generation from Labs 1-5.

Overall System Descriptions, ASMD and Block Diagrams



Figures 1: Showing game layout of Red Light Green Light

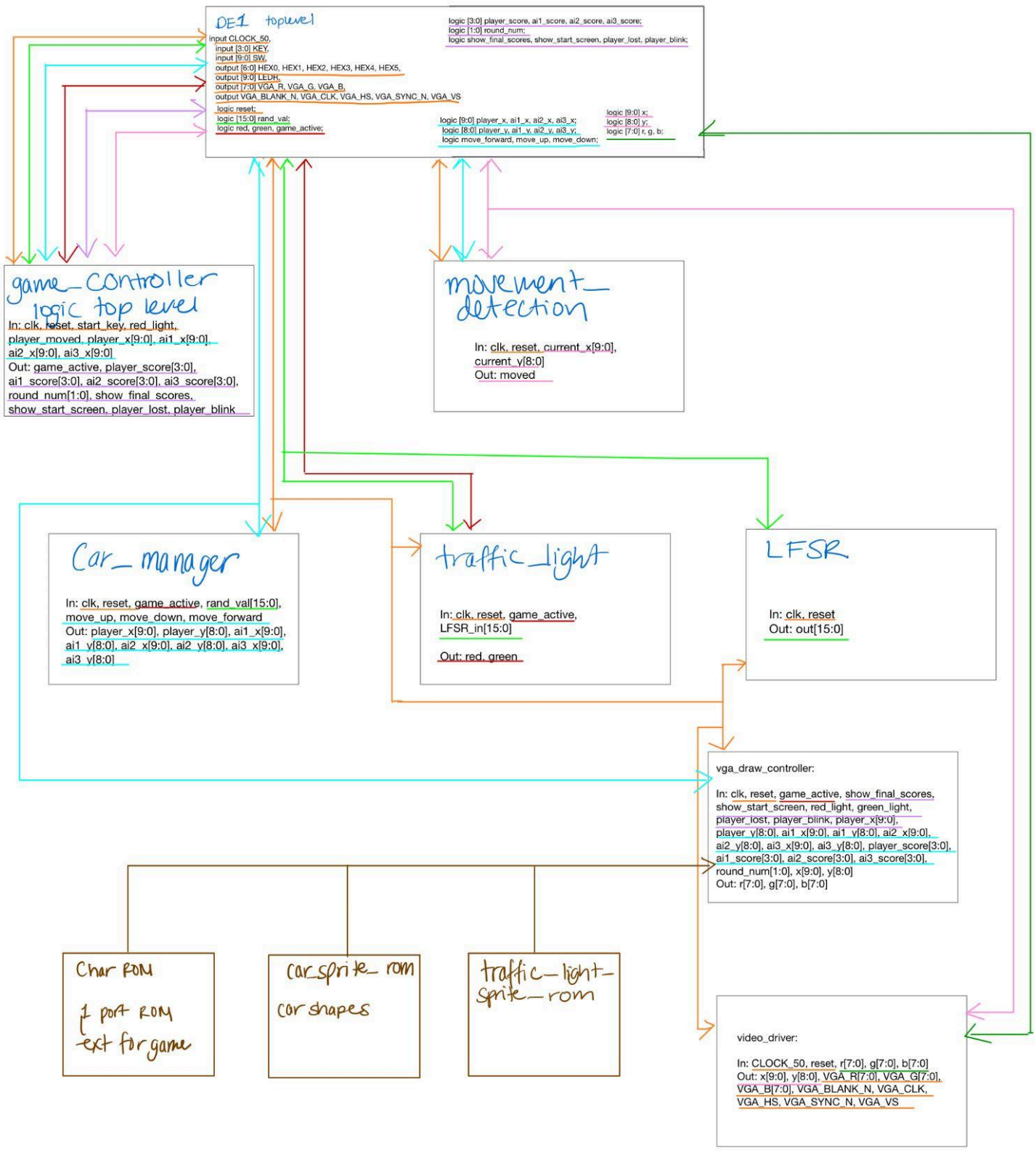


Figure 2: Module Block Diagram for Red Light Green Light

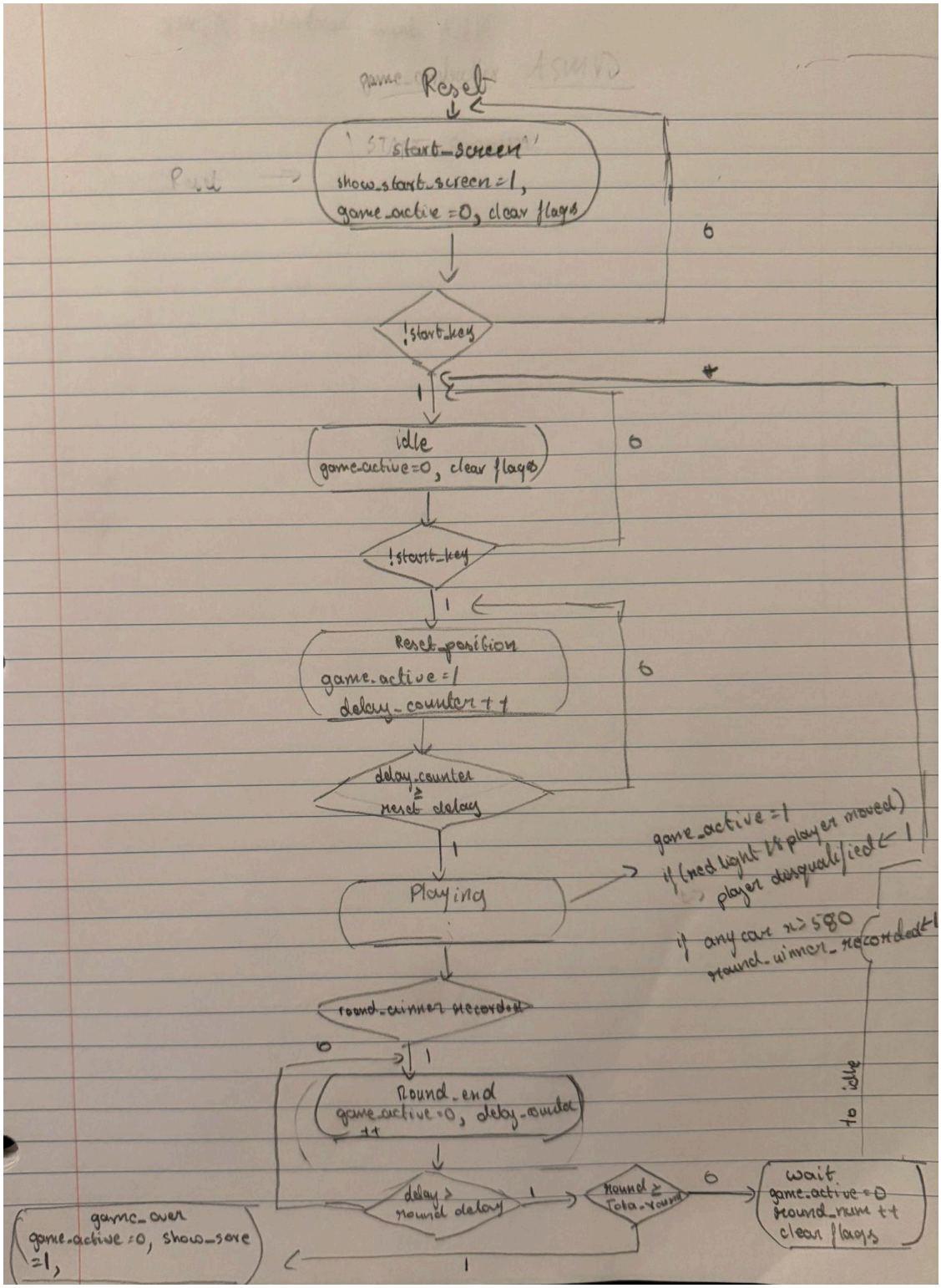


Figure 3: game_controller ASMD Diagram

DE1 Module

DE1_SoC is the top level module connecting the game logic with physical inputs and outputs on FPGA and VGA outputs. Reset KEY[0] clears game state. The lfsr module generates 16 bit random values each clock cycle which is fed to the module traffic_light and module car_manager to make the AI car movement and variable traffic light

durations. The traffic_light receives LFSR output and toggles red/green signals reflected on LEDR[9:8] for visual feedback. The car_manager tracks the player and AI car positions. The player location is updated via board switches (SW[0] forward, SW[1] up, SW[2] down) and the AI uses the LFSR randomness values. The positions are reset at every round start. The movement_detector monitors player_x and player_y and raises a moved signal when coordinates change between cycles to enforce red light rules that user cannot move during red light. It also goes into the game_controller which implements a FSM which controlls game phases: start_screen, idle, reset_positions, playing, round_end, wait_next_round, game_over. It also tracks scores, rounds, enforces disqualification for moving on red and generates the control signals game_active, show_start_screen, show_final_scores, player_lost, and player_blink. LEDR[3:0] shows player score, [7:4] shows AI1 score, HEX5 displays current round, HEX0-HEX4 display scores for player and all AI cars using digit_to_7seg conversion. The video_driver generates VGA coordinates x,y and synchronizes output signals (VGA_R/G/B, VGA_HS, VS, BLANK, CLK) while vga_draw_controller receives x,y and game state to determine RGB values per pixel, and rendering cars, traffic lights, text, and effects. This design is modular so it isolates the randomness generation, movement logic, game logic and display. This makes debugging and expanding easier.

Game_Controller Module

This is the top logic module that brings together all logic components from the modules below. game_controller controls gameplay using FSM with 7 states encoded in 3 bits: start_screen, idle, reset_positions, playing, round_end, wait_next_round, game_over. start_screen waits for player to press start_key (KEY[1]), clears scores, and initializes round number. idle holds until start_key pressed to begin new round. reset_positions briefly moves cars back to starting line while delay_counter counts to reset_delay, preventing false finish detection. playing is the actual gameplay, monitors player_moved against red_light to disqualify if illegal move occurs, and checks player and AI positions against finish_line to assign points, round_end holds for round_delay cycles to display round winner. wait_next_round increments round_num and resets flags. game_over displays final scores indefinitely until reset.

An FSM decides race round, moving violation detection, scoring, and display control. The FSM state encoding uses [2:0] for state_t enum, and transitions occur on rising clk edge w/ synchronous reset overriding logic. An combinational always_comb block computes next state based on current state, start_key debounced input, round_winner_recorded flag, and delay_counter variables. Sequential always_ff block updates ps and manages registered outputs and counters.

To start the game, we listen for KEY[1]. To keep track of scores, a comparator check player_x and ai, N_x inputs against finish_line parameter (10'd580). A priority encoder awards a point to the first player crossing threshold aka finishline, as long as the player_disqualified flag is low at time of crossing (meaning the player did not move on

red). Disqualification detection uses combinational condition (red_light && player_moved) and once triggered that flag persists until round ends. The round_winner_recorded SR latch prevents multiple points awarded in 1 round. Timing uses a delay_counter (27-bit), reset uses reset_delay (27'd2_500_000) for position settling, round end pause uses round_delay (27'd100_000_000) for the display. There is a blink generation for disqualified player which lets the user visually see they've been disqualified, this uses clock division for the effect: condition (delay_counter < 27'd25_000_000) && (((delay_counter / 27'd12_500_000) % 2) == 0) and this creates 0.5-second 2Hz blink pattern.

Output are as follows:

- game_active = (ps == reset_positions) || (ps == playing) enables car motion logic
- show_final_scores = (ps == game_over) activates VGA display
- show_start_screen = (ps == start_screen) enables title
- player_lost directly wires to player_disqualified register
- Score registers (player_score, aiN_score) increment only when corresponding entity first crosses finish_line during playing state with winner flag clear

Round counter round_num increments in wait_next_round state when round_num < total_rounds. The game is ended when round_num >= total_rounds, we chose 3, which transitions to a game_over state and a hardware reset (reset signal from KEY[0]) asynchronously clears registers and returns FSM to start_screen.

We use counters for the timed delays, parameters for finish line and round and flags round_winner_recorded and player_disqualified to prevent multiple scoring or repeated disqualification. An LFSR input from traffic_light controls randomized light durations.

VGA_draw_controller

This module figures out what color each pixel should for the video driver as it scans across the screen. The video driver, provided by teaching team, gives us x and y coordinates for current pixel and this module responds with rgb color values telling what to draw there. There are 3 screens it can render: a title screen with red light green light title text and press key 1 instruction telling player how to start the game. A second screen shows the actual racing game mode with gray road, lane markers, checkered finish line, traffic light sprite and 4 colored car sprites. The last screen displays game over with the winner name and final score plus restart instructions.

For text rendering, this module talks to char_rom which stores 8x8 pixel bitmaps for every letter. We figure out which character we need based on x position and which row of that character based on y position, then we ask char_rom for an 8 pixel pattern and the character rom returns a byte where each bit tells if that pixel is lit up or not. To adjust the size of things, text for example, we scale bigger by dividing coordinates which makes 2x size letters for the title screen.

For car sprites, we use car_sprite_rom which holds 30x30 pixel car shapes. If when scanning the current scan position falls inside a car bounding box we will calculate offset into sprite and ask the rom if that pixel should be solid or transparent. If solid we color it based on which car: player green normally or red when disqualified, ai1 cyan, ai2 yellow, ai3 magenta. This is how the cars render and get their shapes. Also, player car has special blink logic where it alternates visible and invisible when caught running a red light.

The traffic light uses similar sprite system but with 40x40 pixels, the sprite defines two circle areas, top for red light and bottom for green light. When red_light signal is high, top circle draws a bright red circle. When green_light high the bottom circle is lit bright green. Border and spacer areas are white and the background is dark gray.

For the road surface we just checks y coordinate, if between 50 and 480 we draw gray for the rode. Lane markers check if x = 200 or 400, then use modulo math on y coordinate to create dashed white lines for the lanes. The finish line at x 590 alternates red and white stripes using division and modulo to make checkered pattern like a finish line style. The whole thing works combinationally with no delays so when the video driver asks for a pixel we can immediately calculate the color based on our game state and coordinates. This lets us draw an entire frame without needing framebuffer memory, or using the line algorithm from lab 5, which we tried to use initially but was much more difficult. The big advantage here over the framebuffer approach in lab 5 is we never store pixels anywhere, just calculate colors on demand as video beam sweeps. While with framebuffer we had to write pixels into memory using bresenham line drawing, super complicated and caused all kinds of issues like trails and tails when things moved because old pixels stayed in memory and were hard to clear at the proper times. Here we just check if current x y position falls inside something we want to draw which is way simpler.

Car_manager Module

The car_manager module is the traffic controller, it keeps track of when the game is running and updates the positions for the player car along with three AI cars. Every time reset goes high, the module sets 4 cars back to starting coordinates on the starting line, near left side of VGA frame so everyone begins lined up and ready. Then movement logic kicks in. Two timing counters slow everything down. The player gets a 22 bit counter that fires every 2500000 cycles and the AI cars share a 23 bit counter that fires every 5000000 cycles. When these counters reach their thresholds the module allows movement update. If player counter expires, the module looks at the move_forward, move_up, and move_down inputs and shifts the player by 5 pixels in which direction is active. Player can only move between y values of 50 and 430 and an x limit of 600 so the car never goes off screen. We save previous state of each button input so we can detect rising edges later if needed. The AI car movement receives a 16 bit pseudo

random value from the lfsr module and each car reads a different 4 bit slice of it, (rand_val[3:0], [7:4], [11:8]). Two bits control horizontal movement, two bits control vertical movement and when counter hits limit each AI car checks its bits. If the horizontal control bits equal 3 the car moves forward by 10 pixels, vertical bits equal 1 the car moves up by 5 pixels and if they equal 2 the car moves down by 5. Other pattern leaves the car still for that update cycle. Each AI car uses its own bit slice and so all three have independent movement patterns that still stay inside their allowed boundaries.

Movement_detector Module

The movement_detector module take inputs from DE1 current_x and current_y each clock and stores both in prev_x and prev_y, registers form 1 cycle memory so module always holds last position. On rising edge, current_x loads into prev_x and current_y loads into prev_y, then checks current_x vs prev_x and current_y versus prev_y. If they are different we set moved variable to,1, high and output that back to DE1. If they match, the moved variable stays 0, low, bc there has been no movement.

Traffic_light Module

This module is an FSM, ps flips when duration_timer hits 0. The module tracks elapsed time with second_counter which increments each clock and rolls over at SECOND, 1 second meaning 1 second has passed so duration_timer decrements until it hits 0. A reset forces ps to red_light, then sets duration and duration_timer to 3, clears second_counter. When the game_active is high, timing logic runs, duration loads from LFSR_in coming from the lfsr module. When duration_timer = 0 by computing LFSR_in %10+1 which gives a random green or red length between 1-10. Next state logic duration_timer hits 0, FSM flips between red_light and green_light. Red output asserts when ps = red_light and green output asserts when ps = green_light. Overall the module behavior is a 1 cycle FSM, a counter for clock to second conversion, a timer for random duration, plus the combinational logic for the states.

LFSR module

The lfsr module generates 16 bit random stream by shifting out right each clock and feeding in a new bit computed from xor of tap positions 15, 14, 12, 3. On reset loads seed 0xACE1 so sequence always starts in a known nonzero state. 0xACE1 is used as seed to initialize the lfsr because it is a nonzero value that ensures lfsr starts in a valid state bc fsrs cannot start at 0 or they will lock and output zeros forever bc of the shifting. 0xACE1 was chosen because it is a common seed for 16 bit lfsrs with taps at 16, 15, 13, 4 and it produces a full 2^16-1 sequence for max random output.
Update logic forms by concatenating lower 15 bits with new feedback bit so sequence advances every cycle, taps form a max length number so cycle covers all nonzero 16 bit patterns before repeating. We used an lfsr because hardware cost stays small: xor gates plus shift register without multipliers, dividers or large lookup memory component so the sequence runs fast and unpredictable.

VGA Video Driver
The module was provided by teaching team. The video_driver generates VGA scan coordinates x,y and synchronizes output signals (VGA_R/G/B, VGA_HS, VS, BLANK, CLK). vga_draw_controller receives x,y and game state to determine RGB values per pixel, rendering cars, traffic lights, text, and effect player_blink. Design modularity isolates randomness, movement, game logic, and display, ensuring deterministic FSM control, responsive input, AI variability, score tracking, and consistent VGA output without blocking logic, while LFSR provides efficient hardware-friendly pseudo-random sequences for dynamic gameplay.

Car_sprite_rom module
The module receives 2 coordinates named row and col that represent pixel position relative to the top left corner of the sprite, an output named pixel_on becomes high when the coordinates fall inside regions. The shape is created through a case statement that evaluates bycurrent row where each row forms horizontal slice of sprite, rows 0-3 output zero for pixel_on because the upper region of the sprite contains space. Once at row 4 23 compare col against small numeric intervals, row 4 turns pixel_on high for col values from ten through nineteen to form a thin roof line. As rows increase, width of the active interval expands to shape the car and later rows make voids for wheel. The body is drawn from bounds on the col signal and are a bitmap mask that VGA can sample in real time and keeps timing simple since renderer avoids performs fixed sets of comparisons/pixel.

Char_rom module - Memory
Memory is used to store all character bitmaps in contiguous block via a 1 port ROM. Each character occupies 8 consecutive addresses one per row. Address computed as {char_code[6:0], row[2:0]} giving ROM index. A MIF file preloads ROM with bitmap patterns for ASCII characters. On clock edge ROM outputs 8 bit row pattern to the pixels. It is a lookup table. Memory usage here is 128 chars × 8 rows × 8 bits = 8192 bits and the access is combinational via address concatenation. The memory is read synchronously on the clock.

Character ROM stores 8x8 pixel bitmap for rendering text for welcome and game over screen. Address port combines 7 least significant bits of incoming character code with 3 bit row index forming 10 bit address selecting 1 of 1024 bytes in memory. ROM is preloaded from MIF file and each byte contains 8 bits corresponding to 8 horizontal pixels on selected row. 1 lights pixel, 0 leaves pixel dark. ROM captures address on rising clock, outputs selected byte next cycle through pixels. Layout stores characters contiguously: character code selects block, row index selects offset inside block. char_rom holds bitmaps, memory stores 8 rows per character, each row as 8-bit wide pattern, each bit drives pixel on/pixel off.

We chose this because it is simple and efficient and does not take heavy computation. Using a MIF we preload the character bitmaps which avoids runline calculations and big arrays. It is low resource and fast access.

Traffic_light_sprite_rom

Traffic light sprite ROM builds a 40x40 image using comb on sprite_x and sprite_y. pixel starts at 0. Case on sprite_y controls all geometry, rows 0 and 1 force pixel to 1 for a solid top border. Rows 2 and 3 draw side edges by setting pixel to 1 when sprite_x <= 1 or sprite_x >= 38. Rows 4 through 15 form the red lens, side edges always draw, inner fill uses row specific ranges. Early rows use 12- 27, middle rows expand to 7 - 32. Later rows again to 12 -27, producing a circular shape. Rows 16 through 18 draw only side edges for a spacer, rows 19 - 30 form the green lens using the same pattern, 31 - 37 revert to side edges only, 38 and 39 force pixel to 1 for bottom border. Module uses no memory so pixel value read instantly.

Results and Testing

Overall, our final product was identical to our proposal minus the implementation of moving obstacles which we decided not to implement as it complicated the logic beyond what we realistically had time for in 2 weeks.

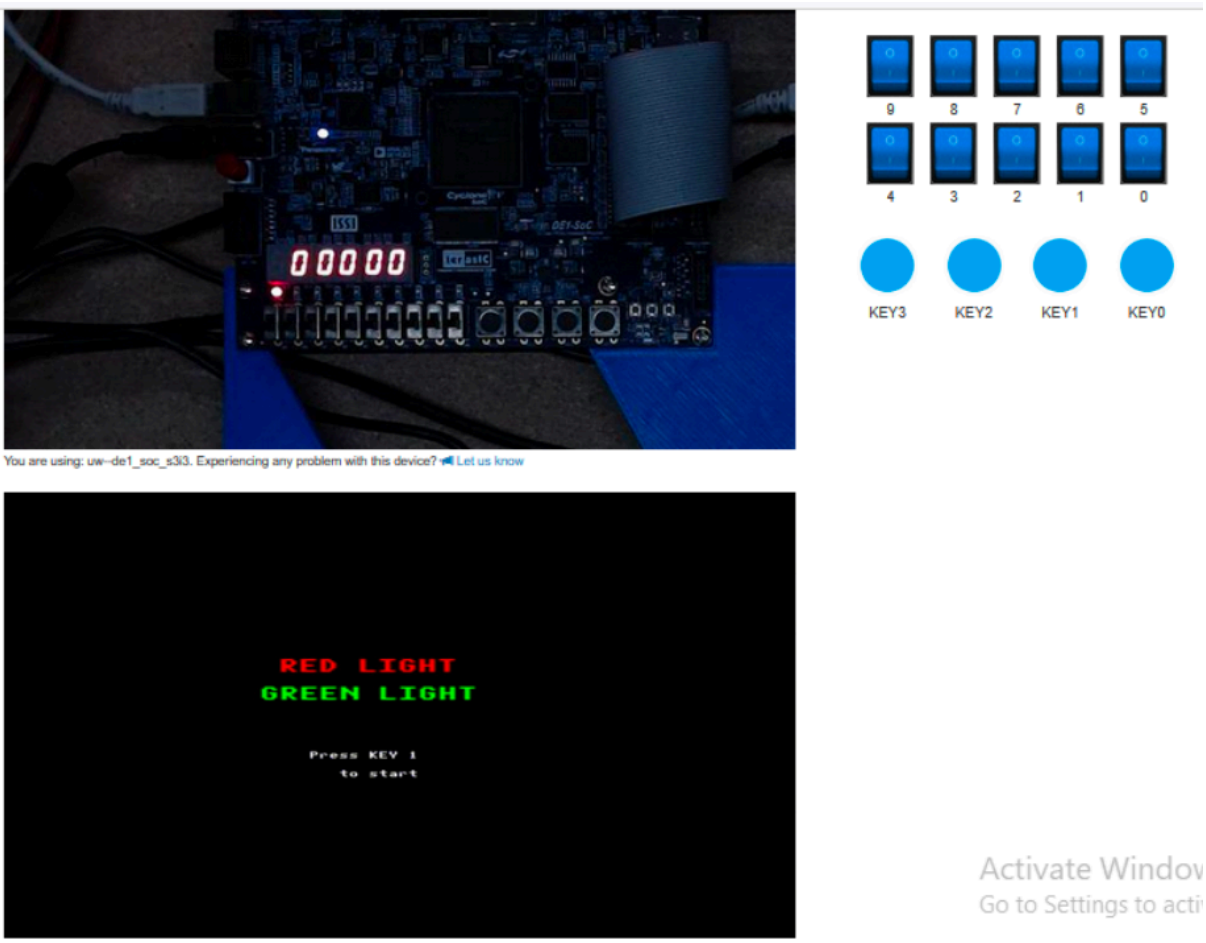


Figure 4: Game welcome screen and reset screen

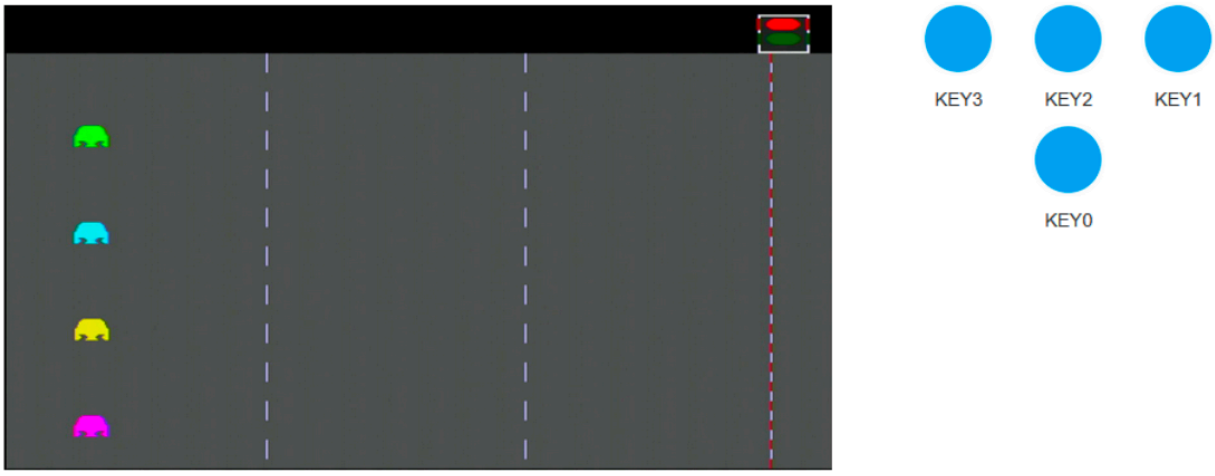


Figure 5: Starting positions for cars after pressing KEY[1]

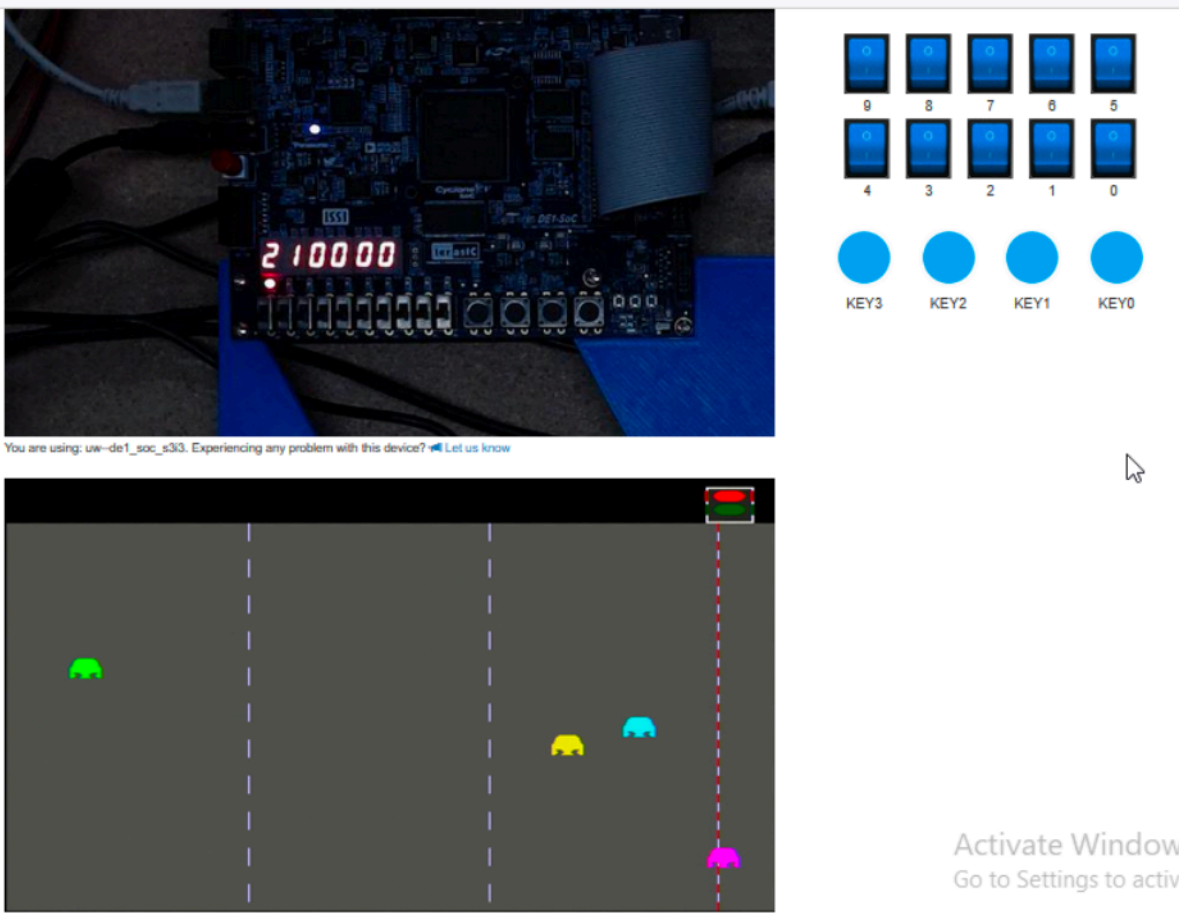


Figure 6: Showing AI1 winning, incrementing score on left hand side to 1 and round to 2

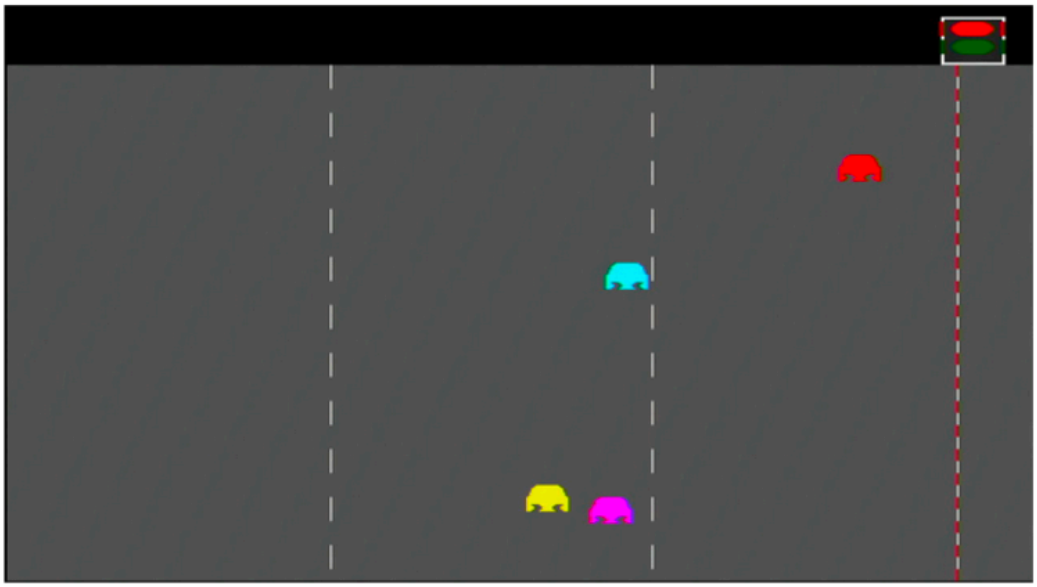


Figure 7: Player car turning red when moving on red light

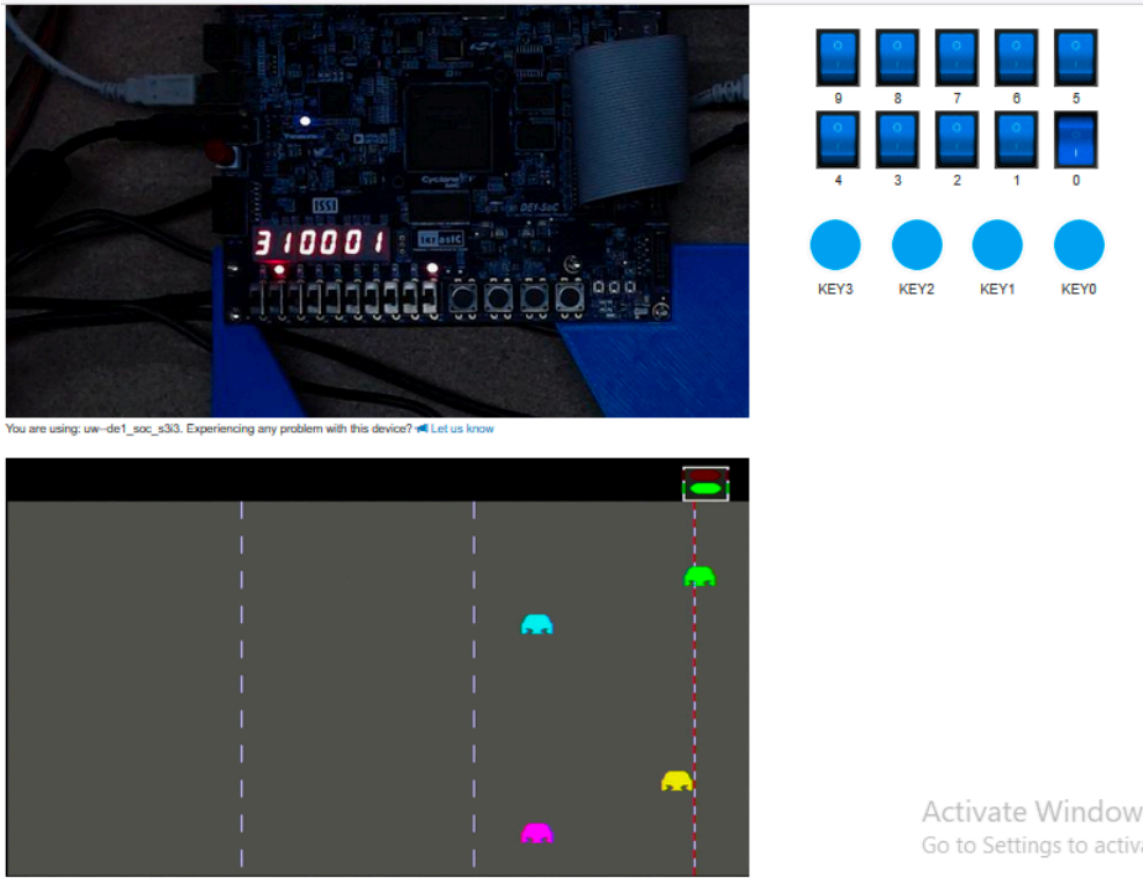


Figure 8: Player moving on green light, crossing the finish line with round point incremented on right HEX display, round is incremented to 3

Figure 9: Game over screen showing user won

Figure 10: Game over screen showing AI car 2 won

tb_DE1_SOC:

We did not create a test bench for DE1 module because it only connects modules together and contains no actual logic in itself. All control, timing, scoring, and game behavior live inside top logic module, which is game controller, so testing focused on that instead. We were instructed this was fine.

tb_game_controller:
This testbench checks states, scoring rules, red-light enforcement, and round progression for correctness.This testbench tests game_controller FSM and scoring logic. It verifies game_controller manages complete three round racing game correctly. clk toggles every 10 time units, reset initializes game state. start_key, red_light, player_moved, and player_x/ai_x signals emulate player inputs, traffic light status, and car positions. DUT instance connects these signals to outputs game_active, player/AI scores, round_num, show_start_screen, show_final_scores, player_lost, player_blink. Test sequence begins by asserting reset then verifying start_screen active, pressing start_key transitions FSM to idle and then reset_positions, waiting full reset_delay cycles so cars ready to start a racing. Test 1 confirms correct behavior of title screen. Test 2 simulates starting round 1 and ensures game_active high and round_num correct. Test 3 moves player to finish line through green light making sure player_score increments +1 and FSM goes into round_end. Test 4 simulates player moving during red light in round 2 by checking player_lost flag triggers disqualification, which ensures player cannot earn points when crossing finish line after disqualified. It also checks to see if AI1 earns a +1 score correctly. Test 5 simulates round 3 with AI2 winning, waits round end, and asserts show_final_scores is active with final round_num being correct.

- Test 1: starting from title screen
    - Verifies game powers up in start_screen state with show_start_screen signal high, important because players need to see instructions before game begins
- Test 2: starting round 1
    - Presses start_key button twice to transition from title through idle into first round then waits for reset_delay cycles while cars return to start position
    - Checks game_active goes high and round_num shows 1.
    - Tests button input handling and state transitions work correctly. Without proper state flow it would get stuck at title or skip straight to gameplay without resetting positions
- Test 3: player wins round 1
    - simulates player car crossing finish line at x=590 during green light with player not disqualified.
    - Checks player_score increments + 1 point.
    - Waits for round_delay showing results screen
    - Verifies finish line detection works and scoring system works
- Test 4: player disqualified in round 2
    - sets red_light high and player_moved high simultaneously to trigger cheating detection.
    - Checks player_lost flag goes high marking disqualification then moves player car past finish line but verify player_score stays at 1 without adding point. Moves ai1 car across finish instead and confirms ai1_score increments to 1.

    - Tests critical rule enforcement where moving during red prevents winning even if player crosses finish first
- Test 5: round 3 and game over
    - completes final round by having ai2 win so checks if ai2_score increments properly and verifies show_final_scores goes high and round_num shows 3 after all rounds complete.
    - Tests game knows when to end and displays winner screen

Assert statements verify correct FSM transitions, score updates, disqualification logic, and display flags. We print debug messages with $display at key points like "Test 1 title screen" which tells what the test is checking and whether assertions passed or failed. Assertions like assert(show_start_screen == 1) else $error("Not showing start screen") don't print unless they fail the test in which case $error prints an error message and highlights the problem which is easy to see in the transcript window.

This testbench test the whole game and every state in order, repeating three times and ending at game_over so sequential progression through all states shows that FSM works correctly and game can complete. The tests scoring logic verifies points awarded to correct winner each round and checks player scoring, multiple different AI scoring, and that a disqualified player gets no points. It shows the main rules are working correctly by checking flags like player_disqualified flag which prevents player from winning after moving on red, the main point of the game. It also tests that rounds are properly counted by checking round_num increments properly and that game_active toggles between rounds and positions reset to start before each race. It also tests input handling by simulating button presses for starting game and starting the rounds. It verifies scores are tracked independently for each racer.

Note, this testbench takes ~15 min to complete.

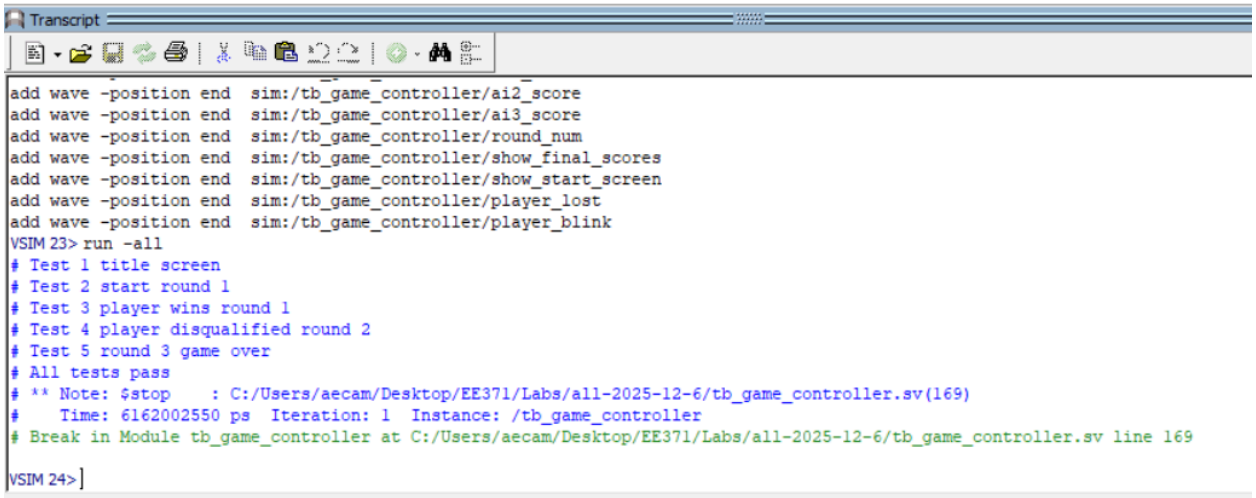


Figure 11: Transcript window showing output of all tests, all tests passed

Waveforms show reset initializing all outputs, then show_start_screen high until start_key press, game_active low until playing state, round_num incrementing after each round, player_score and AI scores updating at finish, player_lost high when player moves on red, player_blink toggling briefly on disqualification, show_final_scores high after last round. Delay counters increment 1-3 after rounds during pause states.

The waveforms are as follows, please read figure descriptions for waveform breakdowns:

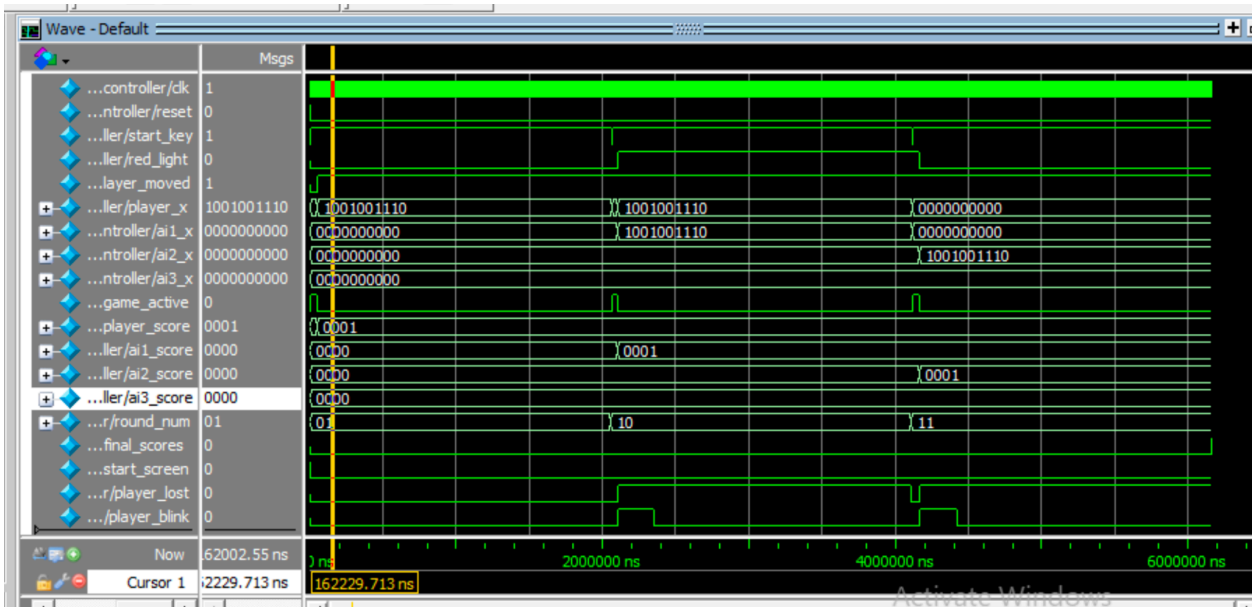


Figure 12: Test 1 & 2: We can see start screen high at beginning after start deasserted (active low), we can see that the round number is one and that no one has scored yet. We can see the position of player x change, but the ai's stay still.

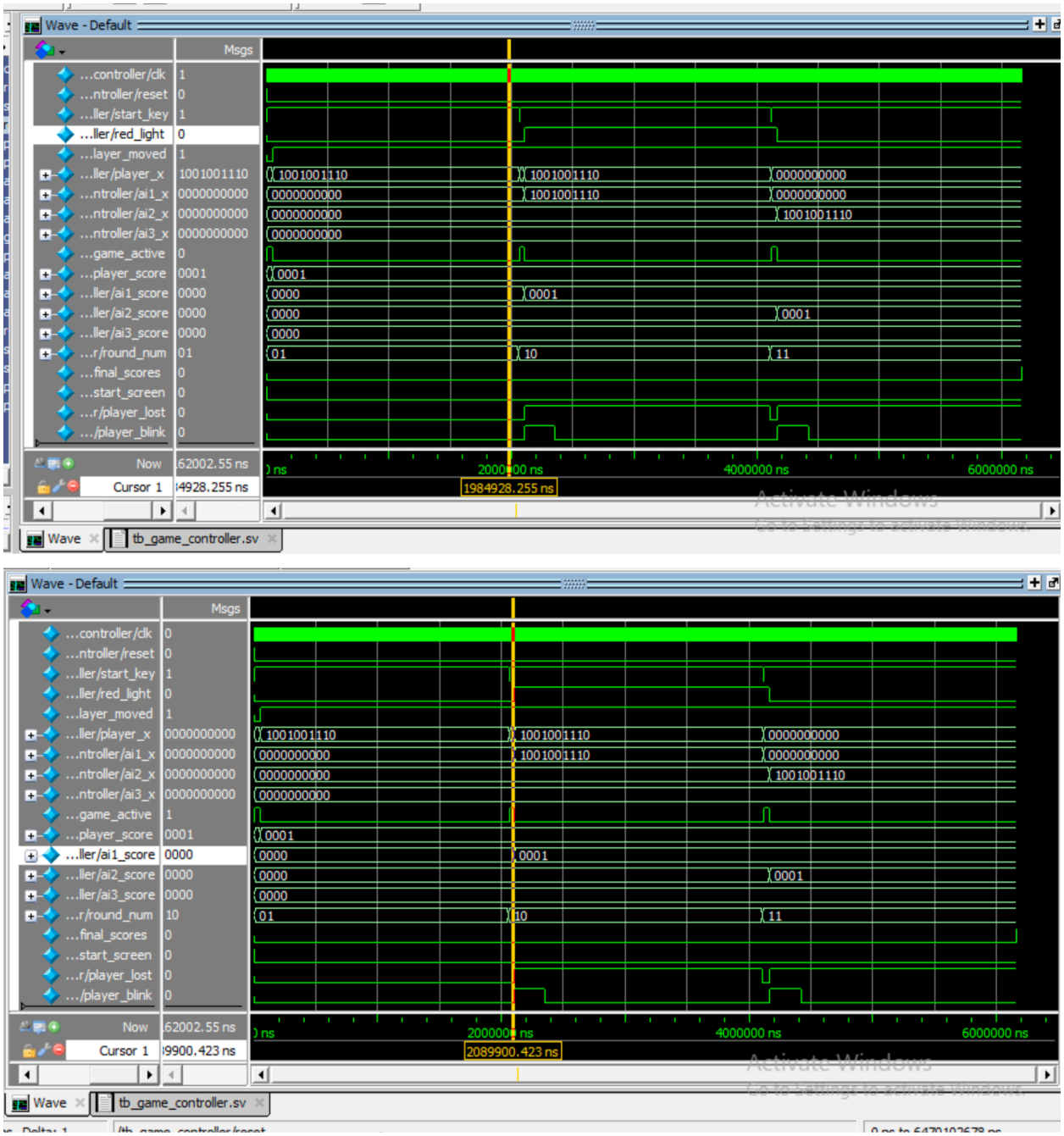


Figure 13: Test 3 &4: Before transition into round 2. In round 2 (near yellow line) we can see the player has 1 point. We can see the positions of player x and ai1 change. Then we can see player continues to move during red light which disqualifies player so player blink effect goes high and player lost flat goes high to 1.

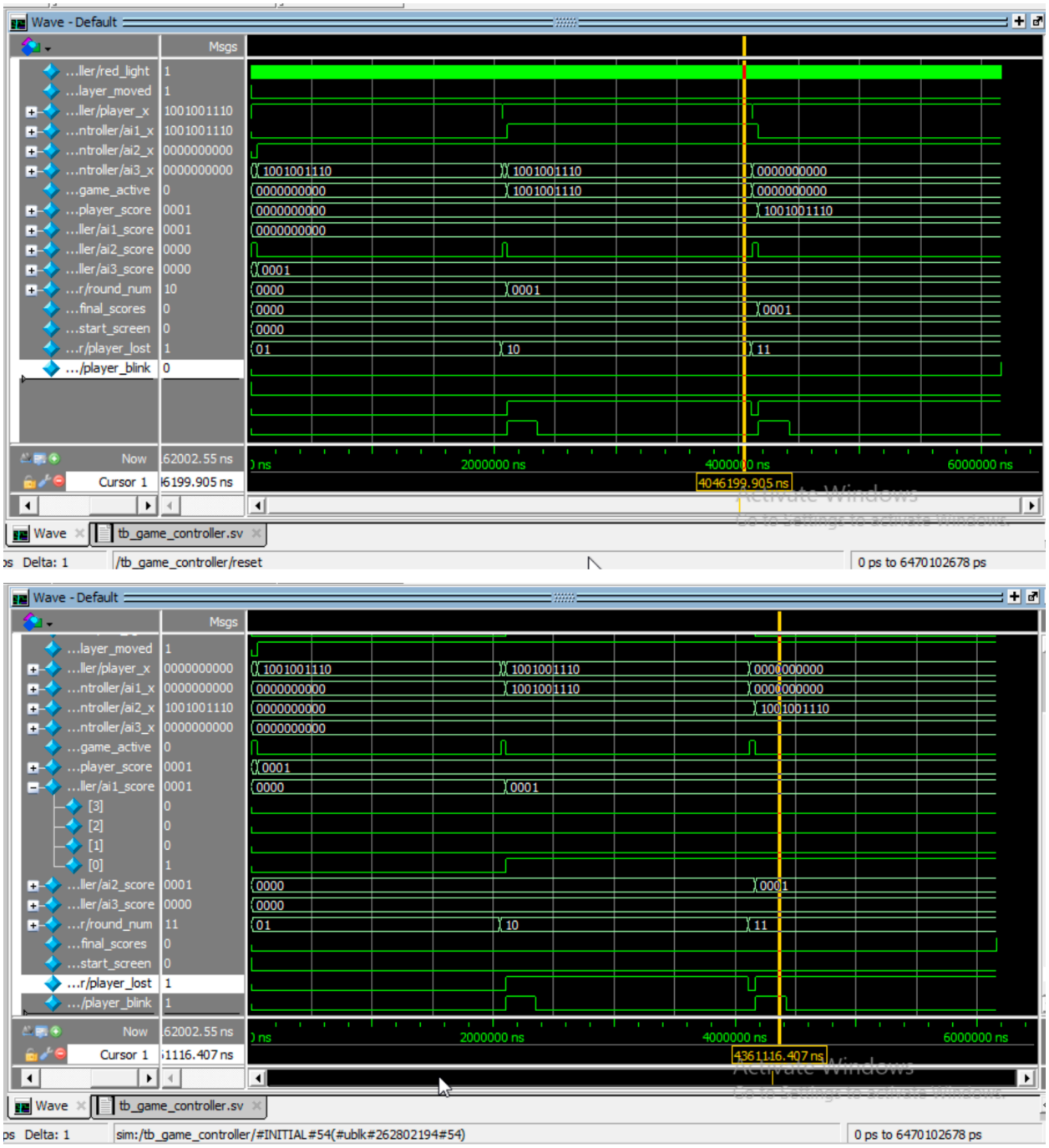


Figure 14: Test 5 Round 3. Completes final round by having ai2 win so ai2_increments to +1. To the right of the yellow line all the way at the end of the line, the final scores screen goes high.

tb_movement_detector

This testbench verifies movement_detector registers the position changes correctly.

- Test 1: no movement
    - Keeps current_x and current_y constant at same values. Checks moved stays low because position not changing. Important because detector should only flag actual movement not trigger falsely when car still
- Test 2: x movement

    - Changes current_x from 0 to 50 pixels. Waits clock cycle for prev_x register to update with old value then checks moved goes high because current_x differs from prev_x. Verifies detector catches horizontal movement car driving forward.
- Test 3: y movement
    - Changes current_y from 0 to 100 while keeping x same. Checks moved goes high from vertical position change and verifies detector catches up/down change
- Test 4: both axes movement
    - Changes current_x and current_y simultaneously, confirms moved still goes high when car moves diagonally. Tests detector works regardless of which coordinates change and if they change at same time
- Test 5: stop moving
    - Holds position constant after movement and verifies moved goes back low once car stops. Important because game needs to know when player holds still during red light versus when moving

Detector compares current position against previous cycle position. When position changes it takes 1 clock cycle for prev registers to store old position. On next cycle, comparison sees mismatch and moved goes high. If position stays same after that, prev catches up to current and moved drops back low. Testbench sequences position changes around clock edges to verify 1 cycle detection delay works.

This testbench effectively tests movement_detector behavior covers all input combinations because it tests every way position can change including x only, y only, both together and neither. It also tests edge cases when nothing moves and when movement stops after starting. And it verifies timing behavior because movement detection relies on 1 cycle delay between current and previous position, so testbench waits for the correct number of cycles after changing positions before checking output.

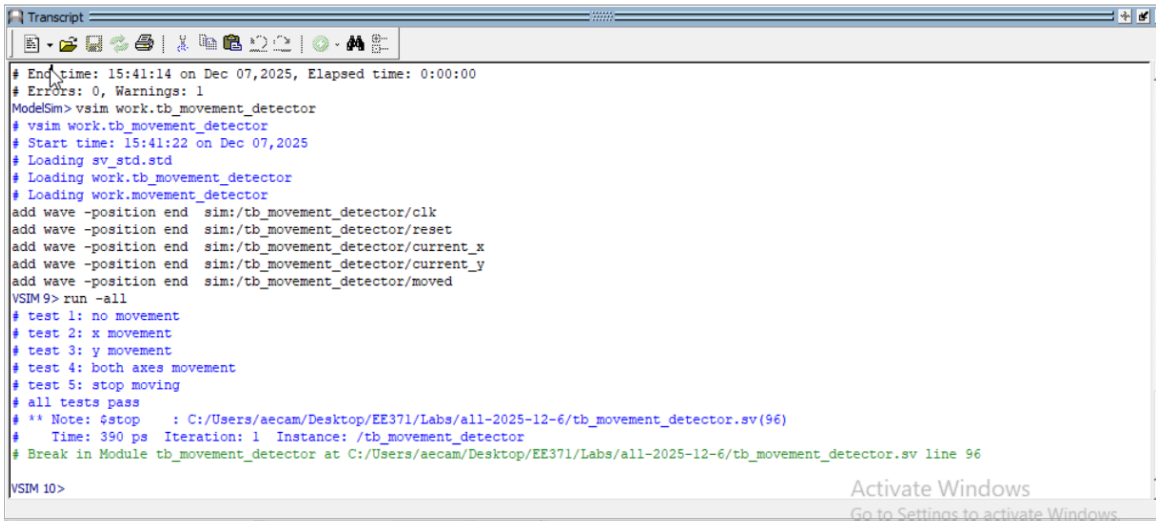


Figure 15: Transcript window showing output of all tests, all tests passed

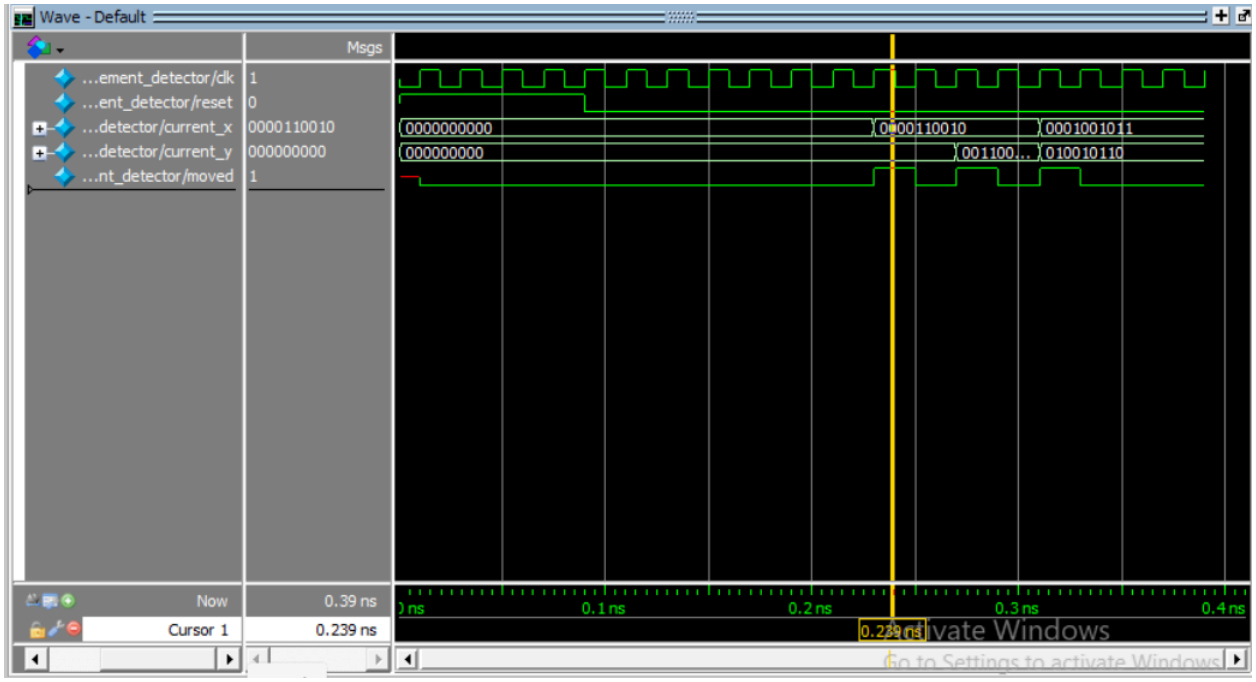


Figure 16: Showing waveform of Test 1, no movement on x nor y thus movement flag stays 0. Waveform of Test 2 at yellow line shows movement in x direction therefore triggering move flag high to 1

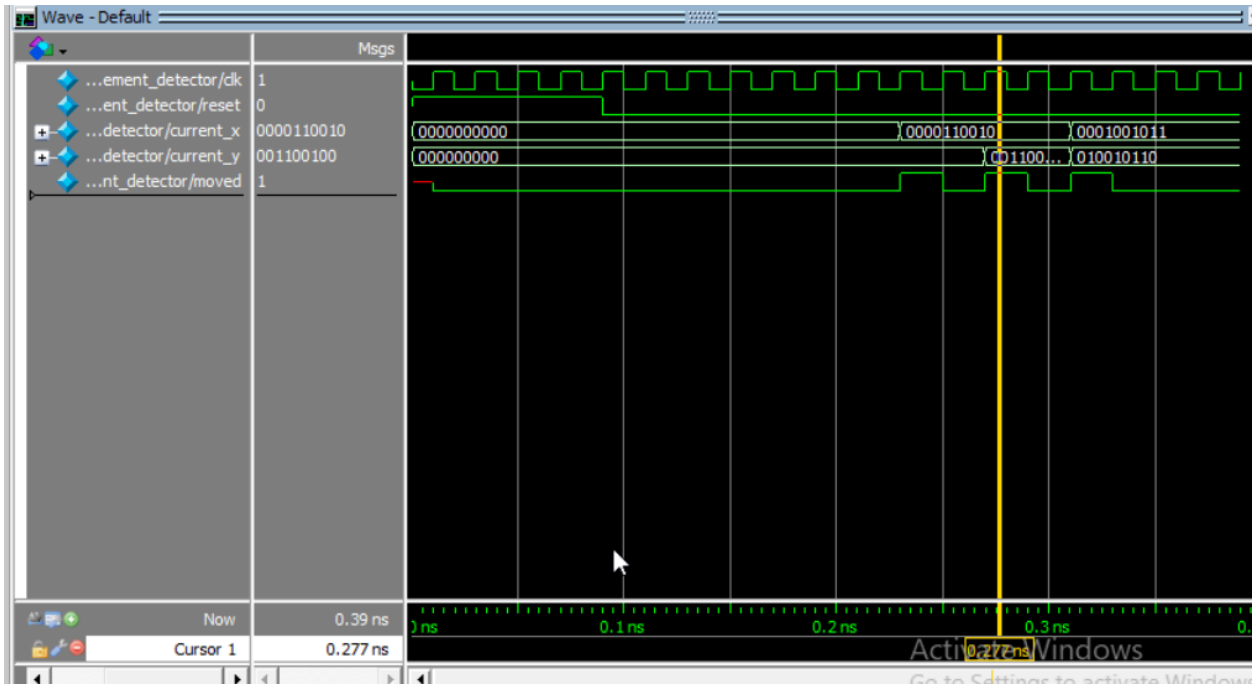


Figure 17: Waveform of Test 3 at yellow line shows no movement in x direction but movement in y direction, therefore triggering move flag high to 1

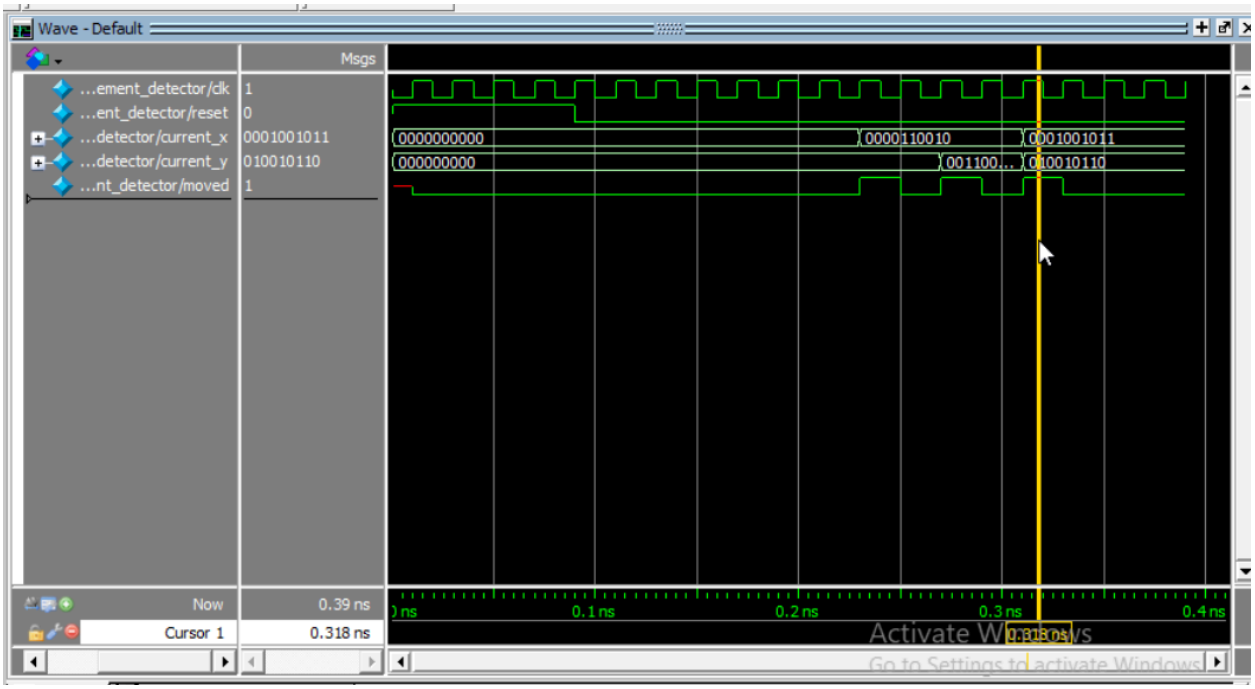


Figure 18: Waveform of Test 4 at yellow line shows both movement in x direction AND y direction, therefore triggering move flag high to 1. Then showing waveform in Test 5, showing no change in movement therefore flag going back down to low, 0

tb_lfsr

We first wrote a 4 bit lfsr and tested the logic on that before expanding it to 16 bit. This testbench initializes the former to verify shift and feedback behavior. Clk toggles every 5 time signals, reset is asserted high initially, forcing LFSR output to known seed, ensuring a deterministic starting state not zeros. After 20 time units reset is deasserted which allows lfsr to shift, on each clock it takes its current output and applies XOR feedback from the specified taps, shifts all bits left, and inserts feedback into the least significant bit. The always block at a posedge clk prints current lfsr generated value each cycle. Waveform shows output at seed, then low then lfsr utput changing each cycle according to feedback logic. It outputs a pseudo random sequence that does not immediately repeat because it cycles through all non zero states before returning to the seed value and follows the XOR tap pattern defined in module.

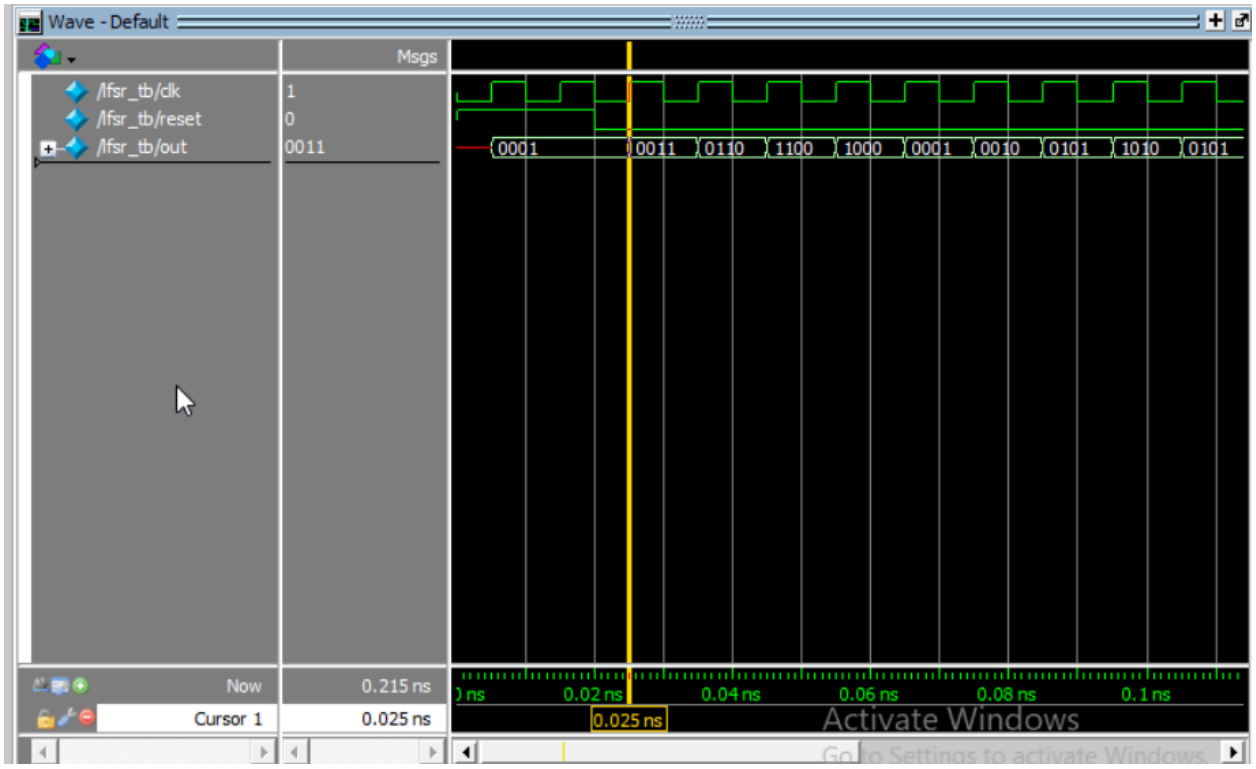


Figure 19: Showing LFSR incrementally shifting

tb_traffic_light

This testbench simulates traffic_light behavior with controlled LFSR input to produce multiple durations. Reset initializes module, game_active starts counting, LFSR_in initially 3, incremented by 7 each time duration_timer reaches 0 and duration changes. prev_duration prevents repeated prints for same duration.

The outputs of 3, 4, 1 means the first light duration was 3, second 4, third 1, reflecting how duration is calculated from LFSR_in %10 + 1. On waveform, red and green toggle based on duration_timer with duration visibly changing at each light switch as seen in the waveforms below. Each $display shows new duration once per cycle, verifying light switches correctly and duration updates with LFSR input.

Note, simulation takes > ~15 min to complete b/c the parameter is in seconds.

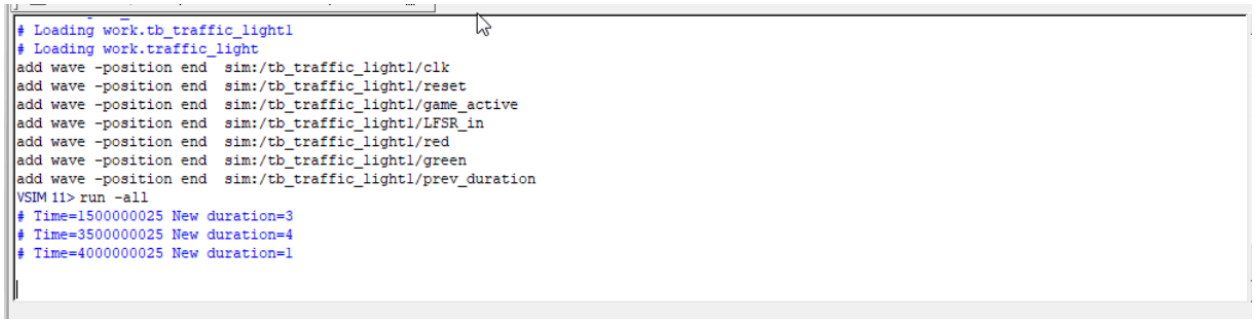


Figure 20: Traffic light using random durations 3,4,1 generated by LFSR

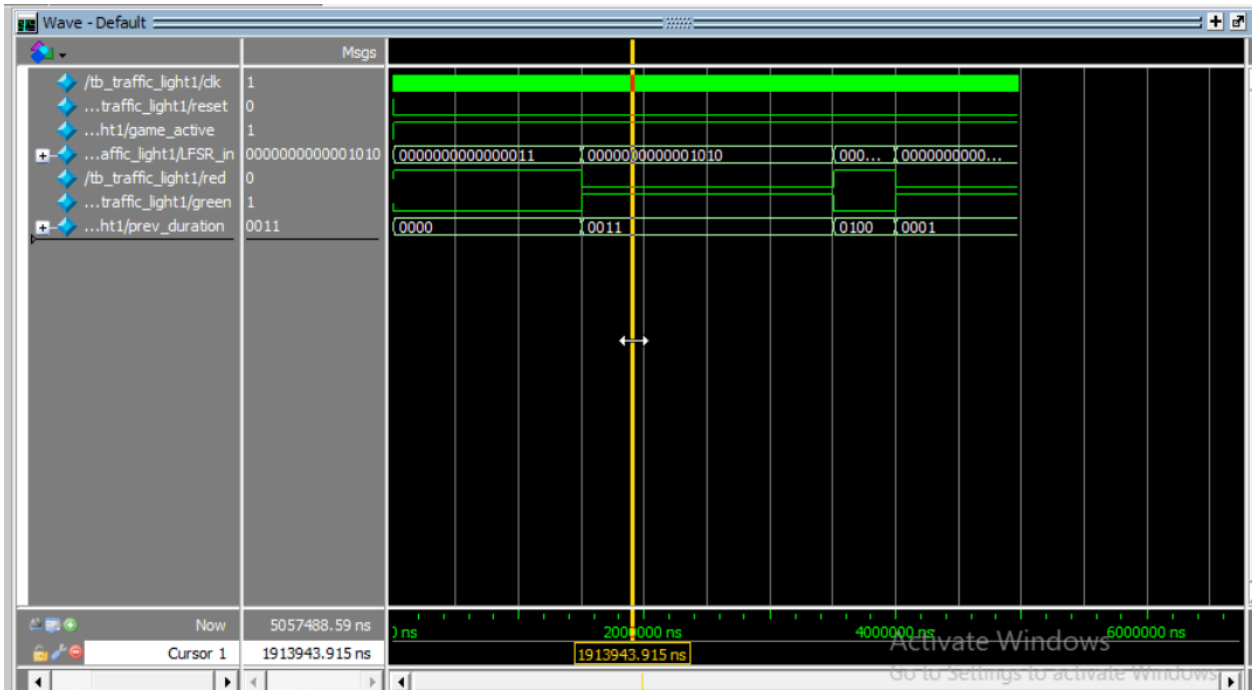


Figure 21: Waveform for traffic light using random durations 3 generated by LFSR, red and green signals alternating and responding and expected

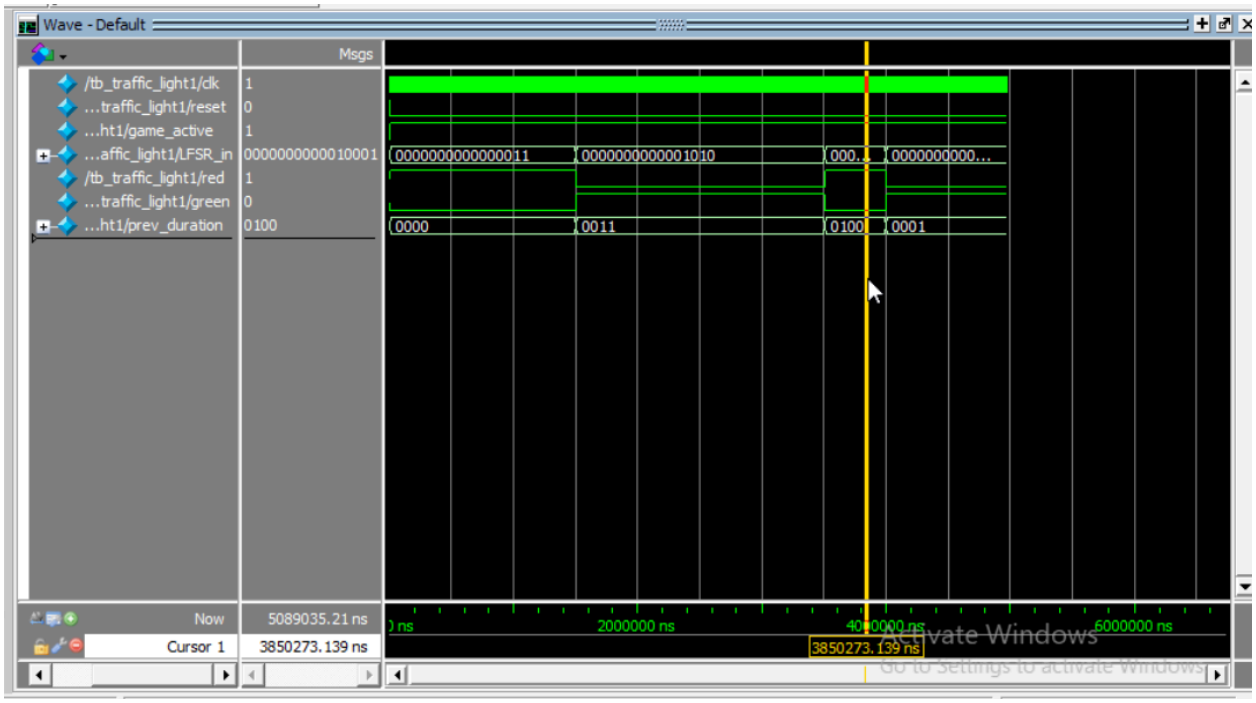


Figure 22: Waveform for traffic light using random durations 4 generated by LFSR, red and green signals alternating and responding and expected

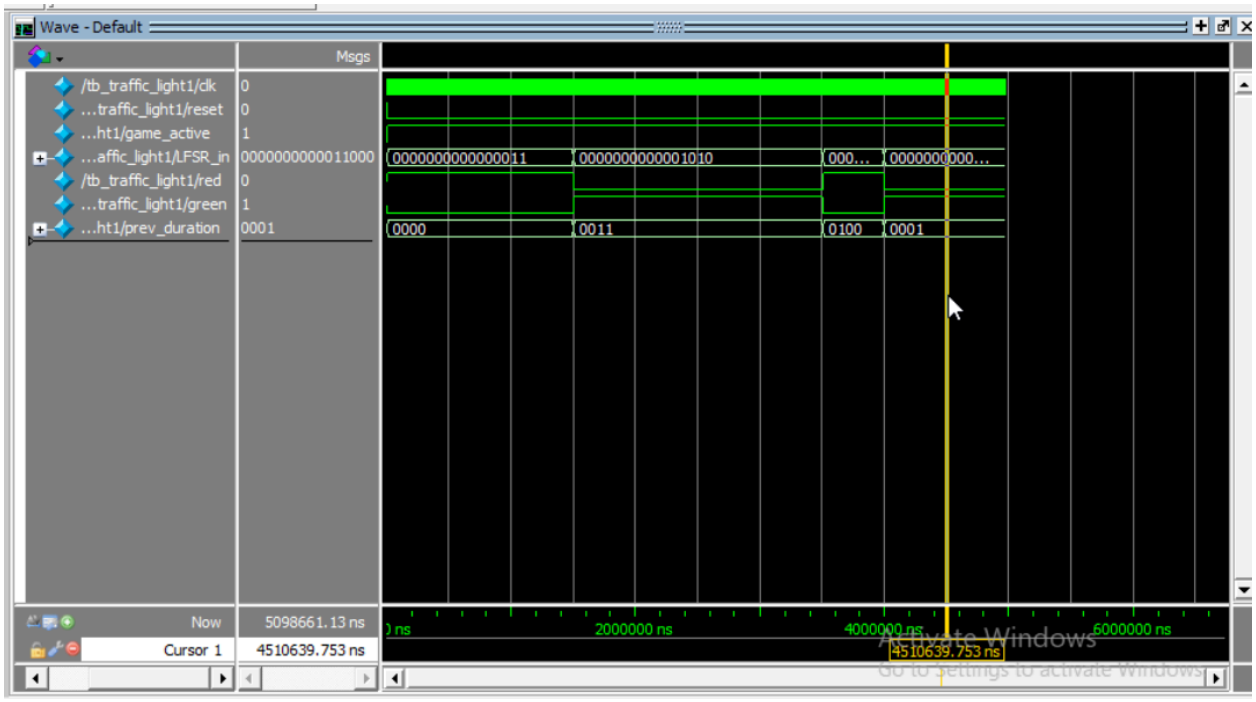


Figure 23: Waveform for traffic light using random durations 1 generated by LFSR, red and green signals alternating and responding and expected

tb_vga_draw_controller

We did not make test bench for VGA drawer control module because waveform test bench works great for verifying control logic like show_start_screen, game_active, red_light, green_light, blink behavior, score updates, car position changes, screen mode transitions. However, pixel visuals stay unreadable since x y sweep across full screen every frame causes RGB signals change nonstop and there are hundreds of them so the waveform turns into noise and would be unrealistic to read, screenshot and follow. We tested visual correctness on actual VGA display.

tb_char_rom

We did not make test bench for char_rom because it's just an initialization of char_rom_qrom, char_rom uses a MIF file to preload font data so correct behavior depends on memory contents rather than logic correctness.

Flow Summary

Flow Summary

Flow Status	Successful - Sun Dec 07 15:21:03 2025
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	N/A
Total registers	376
Total pins	96
Total virtual pins	0
Total block memory bits	8,192
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1
Total DLLs	0

Flow Summary for Red Light Green Light