COL 215 Lab8-part2 : Car Control Logic and Collision Detection

Megh Bhavesh Jesalpura (2024CS10844)
Preesha Ashish Agrawal (2024CS50888)

Department of Computer Science and Engineering Indian Institute of Technology, Delhi

November 7, 2025

1 Objective

The main objective of this part of the lab was to implement control logic for the car sprite displayed in Part 1. This involved using the on-board push-buttons (BTNL, BTNR, BTNC) to move the car horizontally, detect collisions with the road boundaries, and reset the car's position. This control logic was implemented using a Finite State Machine (FSM). A secondary objective was to simulate road movement by creating a scrolling background.

2 Lab work

We were required to modify the existing Display_sprite module from Part 1 to include new logic. This logic reads input from the Basys3 push-buttons to update the car's horizontal position (car_x_current) and manage the game state (e.g., COLLIDE, IDLE). We also implemented a vertical offset counter (disp) to modify the background's ROM address, simulating downward motion.

2.1 Design Decisions

The primary design decisions involved the structure of the FSM to manage the car's state, the logic for car movement, and the implementation of the scrolling background.

2.1.1 Files involved and explanations of modifications

- **Display_sprite:** This top module was significantly modified. It now takes BTNR, BTNL, and BTNC as inputs. It contains the core FSM logic with state registers (current_state, next_state), car position registers (car_x_current, car_x_next), and two counters (cnt, cnt2) for movement speed and background scrolling, respectively.
- **VGA_driver, Horiz_counter, Vert_counter, clk_divider:** These underlying modules from Part 1 were used as-is, with no modifications, to continue driving the VGA display.

2.1.2 File Modifications: Technical Implementation

Instead of a single code block, we provide a detailed breakdown of the logic implemented within the Display_sprite.v module.

State Parameters and Module Instantiation The module instantiates the required VGA_driver and the two ROMs (bg_rom and main_car_rom) from Part 1. For the FSM, five 3-bit localparam values define the states: START = 0, RIGHT_CAR = 1, LEFT_CAR = 2, COLLIDE = 3, and IDLE = 4.

Background Scrolling Logic To simulate a continuously moving road, a vertical offset is applied to the background ROM address. This is achieved using:

- **frame_clean:**A single spike waveform to signal the display of the last pixel in the image, i.e a clean signal when the whole frame has been displayed, created using combinational login on the original and delayed signal.
- **frames_30:** A reg which increments its value upto 4 and back to 0 when a frame display is done, i.e. frame_clean gives a spike.
- **disp_bg:**The offset by which the background has to be scrolled down, incremented after frames_30 reaches a value of 4.

This disp value is then used in the BG_LOCATION always block. The address for the background ROM is calculated by subtracting disp from the current vertical pixel coordinate (ver_pix). The % 240 (modulo) operation ensures the address wraps around the background's height, creating a seamless, repeating scroll.

```
always @ (posedge clk) begin : BG_LOCATION
    if (hor_pix >= 0 + OFFSET_BG_X && hor_pix < bg1_width + OFFSET_BG_X && ver_pix >= 0
    + OFFSET_BG_Y && ver_pix < bg1_height + OFFSET_BG_Y) begin
        bg_rom_addr <= (hor_pix - OFFSET_BG_X) + ((ver_pix - OFFSET_BG_Y - disp_bg) %
    240)*bg1_width;
        bg_on <= 1;
    end
    else
        bg_on <= 0;
end
```

Listing 1: Background ROM Address Calculation

Car Sprite Location Logic An always block (CAR_LOCATION) determines if the car sprite should be active. It checks if the current hor_pix and ver_pix fall within the car's rectangular boundary, which is defined by its top-left corner (car_x_current, car_y) and its width/height. If active, it calculates the correct address for the car1_rom.

Finite State Machine (FSM) Implementation The FSM follows a 'two-always' model to ensure synthesizability and prevent accidental latches. One block is purely sequential for state transitions, and the other is purely combinational for next-state logic.

Sequential Block (State Register) A single always @(posedge clk) block acts as the state register. Its only job is to update the current_state register with the value from next_state on a clock edge. A design decision was to include an asynchronous-style reset. The BTNC input acts as a high-priority reset. If BTNC is high, the block forces the current_state to START (0) and resets car_x_current to 270. This takes precedence over the combinational logic, providing an immediate and reliable reset.

```
always @(posedge clk) begin
    if(BTNC) begin
        current_state <= 0;
        car_x_current <= 270;
    end
    else begin
        current_state <= next_state;
        car_x_current <= car_x_next;
    end
end
```

Listing 2: Background ROM Address Calculation

Combinational Block (Next-State Logic) A purely combinational always @(*) block calculates the next_state and car_x_next. As the block is combinational, all outputs must be assigned a value in all possible paths to prevent latches. We use a case (current_state) statement, and all logic is based only on the current_state and inputs (BTNL, BTNR, cnt). The default assignment next_state = current_state ensures the state remains stable unless an explicit transition is defined.

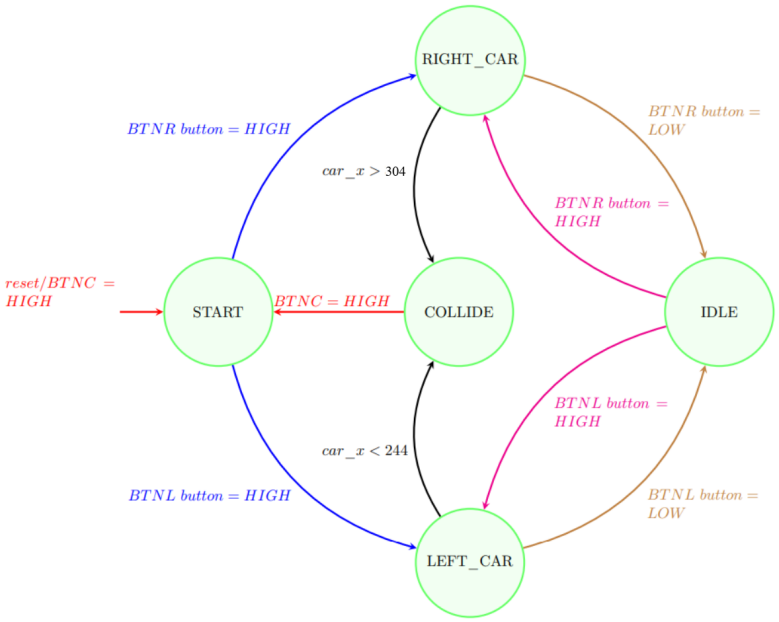


Figure 1: FSM State Diagram for Car Control Logic.

- **START (0) state:** This is the initial state. It sets car_x_next to 270. It then immediately checks the button inputs. If BTNR or BTNL is pressed, it transitions directly to RIGHT_CAR or LEFT_CAR respectively. If no button is pressed, it remains in START (as next_state defaults to current_state).

- **IDLE (4) state:** This is the default stationary state i.e. (car_x_next = car_x_current). The logic monitors BTNR and BTNL. If a button is pressed, it transitions to the corresponding movement state (RIGHT_CAR or LEFT_CAR) and calculates the next position, checking for collision before the transition.

- **RIGHT_CAR (1) state:** This state is entered when BTNR is pressed. A design feature is that the state remains RIGHT_CAR only if BTNR is continuously held down. This allows for continuous movement.
  The logic checks if(BTNR). If true, it stays in RIGHT_CAR (self-loop) and calculates the next movement step.
  If the button is released (else), it transitions to IDLE and stops movement. It also performs a pre-emptive collision check: if the calculated car_x_next would be$>$ 304, it transitions to COLLIDE.

- **LEFT_CAR (2) state:** This state mirrors RIGHT_CAR's logic. It remains in this state as long as BTNL is held, releasing the button transitions to IDLE. It checks for the left boundary collision (car_x_next $<$ 244) and transitions to COLLIDE if detected.

- **COLLIDE (3) state:** This is a terminal 'dead' state. Once entered, all movement logic ceases. The case statement for COLLIDE only checks for BTNC. It ignores BTNL and BTNR, effectively stopping the car. The only exit is by pressing BTNC, which forces a transition back to START via the sequential block's reset logic.

Car Movement Speed Control An important part is decoupling the FSM's state change (which should be fast) from the car's movement (which must be slow enough to be visible). We achieve this using a boolean reg, signal.
i.e. signal is set to 1 when disp_bg is, in the always block that controls it. The car_x_next is incremented by 10 only when signal is on and the FSM is in that state, allowing for sufficient amount of frames to have passed.

This allows for a visibility of the rightward and leftward movement.

3 Pin connections

The implementation now uses the on-board push-buttons. The original VGA and clock pins from Part 1 remain the same.

Port Name	Pin Constraint	I/O
clk	W5	The main 100MHz clock for synchronisation
HS	P19	To represent Horizontal Sync in the VGA
VS	R19	To represent Vertical Sync in the VGA
vgaRGB[11:0]	(various)	Represents the 12-bit RGB ports in VGA
BTNC	**U18**	**Input: Reset button (FSM START state)**
BTNL	**T18**	**Input: Move car left (LEFT_CAR state)**
BTNR	**W19**	**Input: Move car right (RIGHT_CAR state)**

Table 1: Mapping of Ports and Constraints in basys3.xdc (Part 2)

4 Simulation and schematic snapshots

4.0.1 Simulation

We modified the testbench to simulate button presses for BTNL, BTNR, and BTNC. The simulation focused on observing the FSM state transitions, from the start(0) state to a left movement(2), then to the idle state(4) followed by the start(0) state, then to the right(1) and finally collision with the right side(4); and the corresponding update to the car_x_current register.

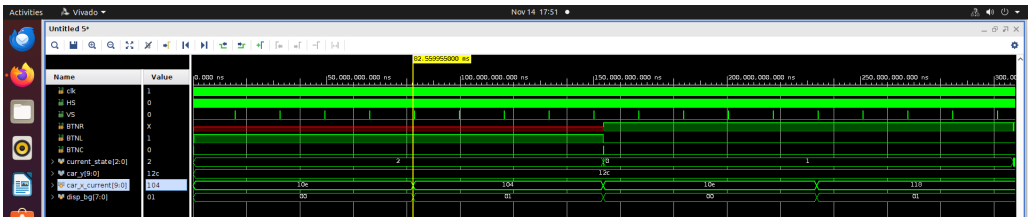


Figure 2: Left Movement at yellow line

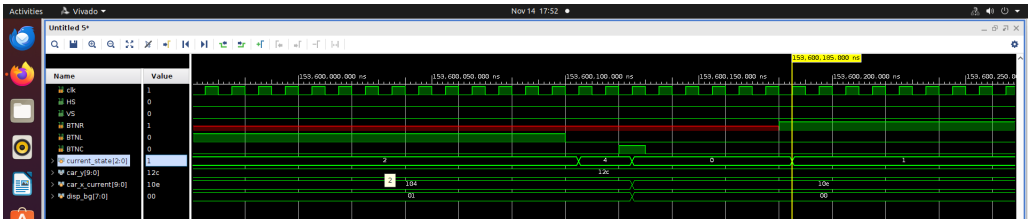


Figure 3: Idle state(4), reset(0) transitioned into

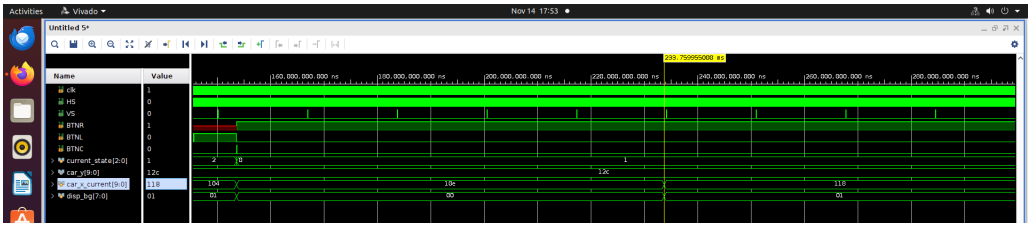


Figure 4: Transition to right state (1), pixel updated right at yellow line

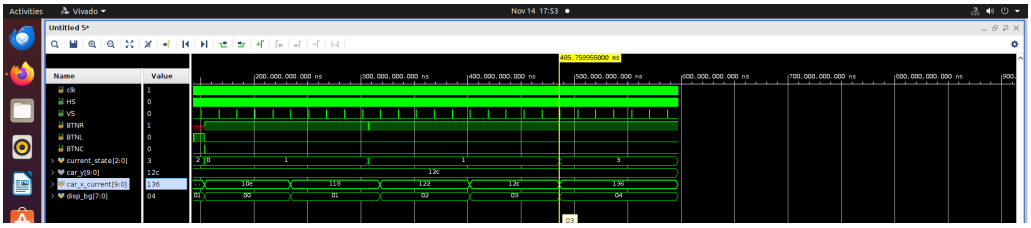


Figure 5: Collision state (3)

When a collision boundary is crossed, the state moves to COLLIDE and both car_x_current and disp_bg reach their maximum value at the same time and stop incrementing, signaling that the background and the car have stopped their respective horizontal and vertical movements.

4.0.2 Utilization report

The utilization of resources increased from Part 1 due to the addition of the FSM, button inputs, and large counters for movement and scrolling, which require more logic and flip-flops.

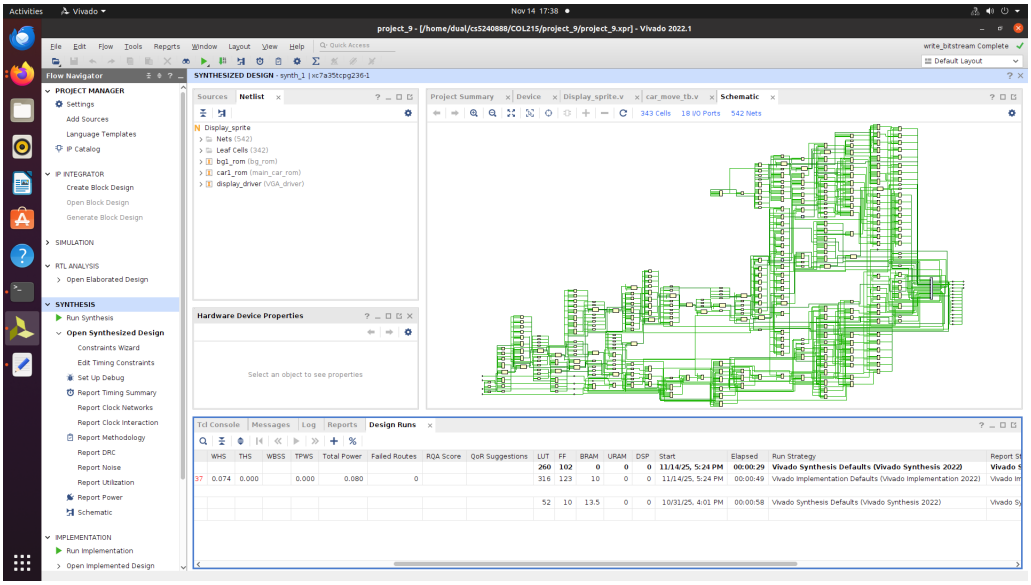
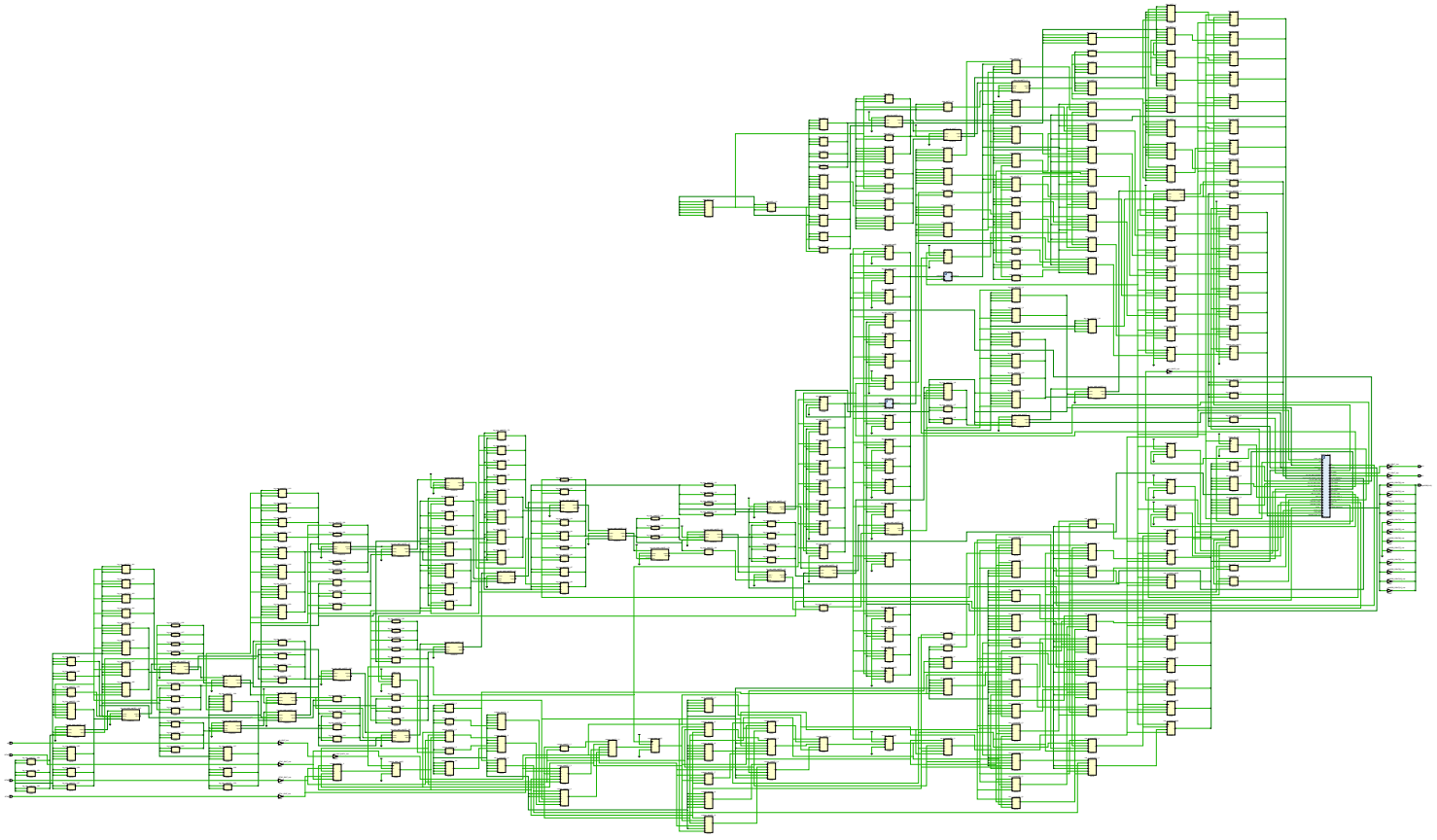


Figure 6: Utilization report

Component	Number of used components
LUTs	316
FF	123
BRAM	10
URAM	0
DSP	0

Table 2: Table of Components and used components



5 Implementation on FPGA board

We uploaded the new bitstream file to the FPGA board. Once programmed, the car was displayed at its initial START position (270, 300).

Pressing BTNR and BTNL successfully moved the car horizontally, and the road was observed to be scrolling downwards. When the car was moved to either edge of the road, it stopped (entering the COLLIDE state), and would only resume after BTNC was pressed to reset it to the START state.

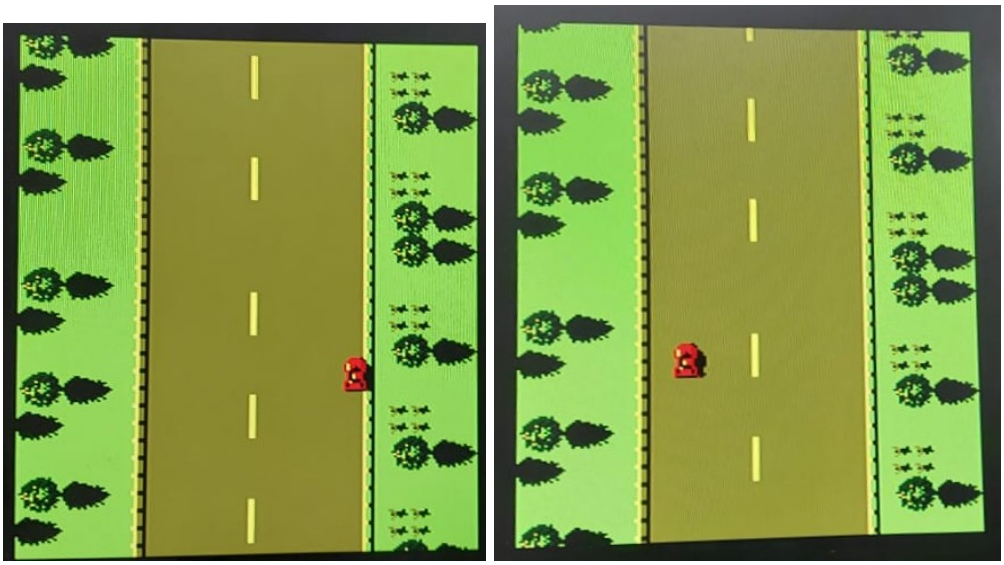


Figure 7: Collision and left movement