COL215 – Digital Logic and System Design

Department of Computer Science & Engineering, IIT Delhi Semester I, 2025–26

Lab Assignment 8 part II (Friday)

Developed By:
Bharmal Bhakar (2024CS10403)
Ritik Raj (2024CS10086)

1 Introduction

Part II of the assignment focused on implementing the control logic for the main car's horizontal movement and collision detection, which is critical for making the game interactive. This was achieved using a **Finite State Machine (FSM)** named Car_control_FSM, which manages the car's horizontal position (current_car_x) based on user input (push buttons btnL and btnR) and road boundaries.

2. Design Decisions

The FSM design follows the standard structure of a State Register (sequential block) and a Next-State/Output Logic (combinational block). The design uses 5 distinct states to manage the game flow: START, IDLE, RIGHT_CAR, LEFT_CAR, and COLLIDE.

2.1 FSM States and Encoding

State	Purpose	Encoding
START	Initial state; sets car position to START_X and transitions immediately based on button press or to IDLE.	3'b000
IDLE	Car is stationary and within road boundaries; awaits user input btnL or btnR.	3'b001
RIGHT_CAR	Car is moving right; continues until btnR is released or collision occurs.	3'b010
LEFT_CAR	Car is moving left; continues until btnL is released or collision occurs.	3'b011
COLLIDE	Game over state; reached when car crosses a boundary; awaits btnC to restart.	3'b100

2.2. Clock Divider for Smooth Movement

To ensure smooth visual movement on the VGA display (which updates at 60Hz), a clock divider was implemented. The movement logic is updated at a rate of 10Hz, allowing the car position to change by a step size MOVE_STEP only when the move_tick signal is high.

- System Clock Frequency: CLK_FREQ_HZ = 100 MHz

- Movement Frequency: MOVE_FREQ_HZ = 10Hz
- Max Count: MAX_COUNT = (CLK_FREQ_HZ)/ (MOVE_FREQ_HZ) - 1
- Move Step MOVE_STEP: 2 pixels (Design Decision for smooth movement).

This ensures the car moves predictably and smoothly, independent of how long the physical push button is held down.

2.3. Collision and Movement Logic

The movement and collision detection logic is integrated within the FSM's sequential block, triggered on the rising edge of the clock clk and guarded by the move_tick signal. The assignment specifies collision boundaries relative to the global monitor resolution (640×480):

- **Start Position (**START_X**):** 270
- **Car Width (**CAR_WIDTH**):** 14 pixels
- **Left Collision Boundary (**COLLISION_LEFT**):** 200 + 44 = 244
- **Right Collision Boundary (**COLLISION_RIGHT**):** 200 + 118 = 318

The collision conditions are implemented using the look-ahead checks on the next potential position. This results in the car halting at 246 on the left and 302 on the right.
If car_x_reg ≤ (COLLISION_LEFT + MOVE_STEP) ⇒ Next State = COLLIDE &
If car_x_reg ≤ (COLLISION_RIGHT - MOVE_STEP) ⇒ Next State = COLLIDE
With MOVE_STEP = 2, this condition is met when car_x_reg is less than or equal to 246. The last safe position is 246 and the last safe position is 302.

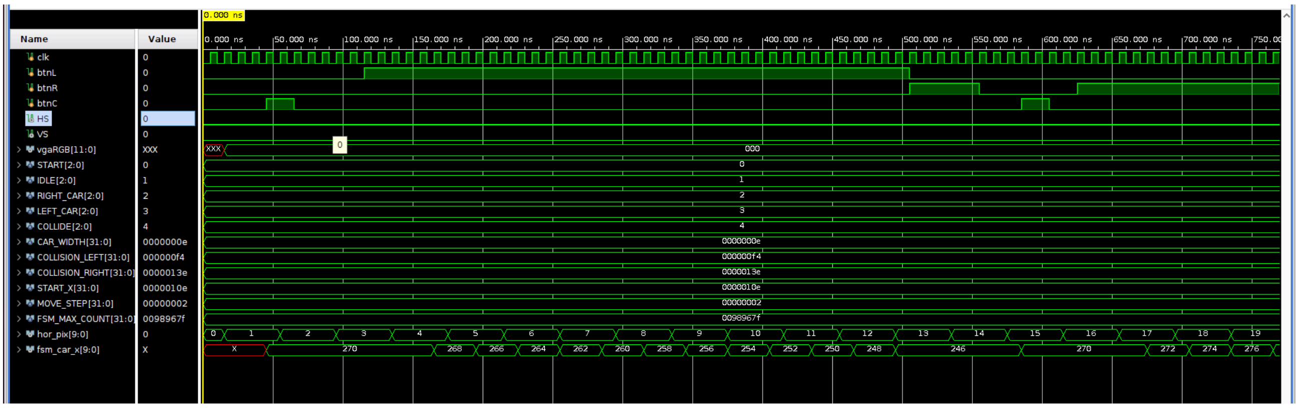
Current State	**Condition**	**Next State**	**Action in Next State (on `move_tick`)**
START	btnL is HIGH	LEFT_CAR	car_x_reg ←car_x_reg - 2
	btnR is HIGH	RIGHT_CAR	car_x_reg ←car_x_reg + 2
	Otherwise	IDLE	car_x_reg ←car_x_reg
IDLE	btnL is HIGH	LEFT_CAR	Move left by MOVE_STEP
	btnR is HIGH	RIGHT_CAR	Move right by MOVE_STEP
RIGHT_CAR	car_x_reg + CAR_WIDTH > COLLISION_RIGHT	COLLIDE	Halt position update
	btnR is LOW	IDLE	Halt position update
LEFT_CAR	car_x_reg < COLLISION_LEFT	COLLIDE	Halt position update
	btnL is LOW	IDLE	Halt position update
COLLIDE	btnC is HIGH	START	Reset car_x_reg ← 270

2.4. TestBench

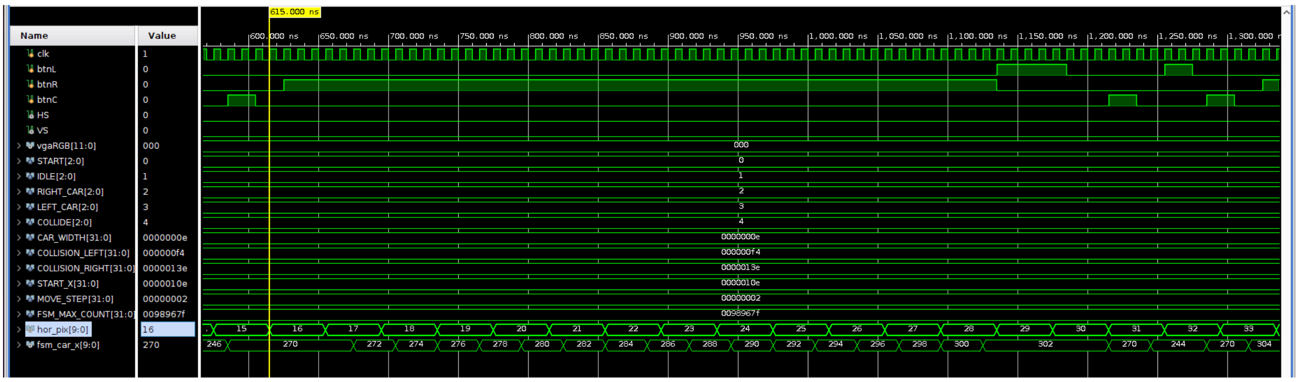
A dedicated test bench was created to verify the functionality of the Car_control_FSM module. The primary focus of the simulation was to validate the FSM's state transitions, the car position updates (current_car_x), and the accurate operation of the clock divider. The test bench was clocked using the 100 MHz system clock, and the simulation was run long enough to observe movement, entry into the COLLIDE state upon boundary crossing, and the reset transition via btnC.

3. Simulation Snapshots

The module was simulated using a test bench to analyze the car position updates (current_car_x). The measured values from the simulation, given a clock period of 10 ns per clock cycle, are recorded and analyzed below:



The btnL signal is held high, causing the fsm_car_x position to decrement repeatedly (e.g., 270 to 268 to 266...) while the FSM is in the LEFT_CAR state (encoded as 3'b011). The car position continues to decrease until it reaches 246. Since the collision condition is car_x < 246, the position halts at 246. At the collision boundary, the FSM transitions to the COLLIDE state (encoded as 3'b100). The car's x-coordinate remains locked at 246, and no further movement occurs, regardless of the btnL state, until the btnC is pressed.



The btnR signal is held high, moving the fsm_car_x position incrementally (e.g., 270 to 272 to 274...) while the FSM is in the RIGHT_CAR state (encoded as 3'b010). The car position continues to decrease until it reaches 304. At the collision boundary, the FSM transitions to the COLLIDE state (encoded as 3'b100). The car's x-coordinate remains locked at 304, confirming that the car is stuck in the COLLIDE state, awaiting a reset from btnC.

4. Synthesis Report

Tcl Console | Messages | Log | Reports | Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSI
synth_1 (active)	**constrs_1**	**synth_design Complete!**												**144**	**120**	**0**	**0**	
impl_1	constrs_1	write_bitstream Complete!	3.791	0.000	0.125	0.000		0.000	0.086	0	49 CW, 3 Warn			193	130	14	0	
Out-of-Context Module Runs																		
bg_rom_synth_1	bg_rom	synth_design Complete!												52	10	13.5	0	
main_car_rom_synth_1	main_car_rom	synth_design Complete!												0	0	0.5	0	

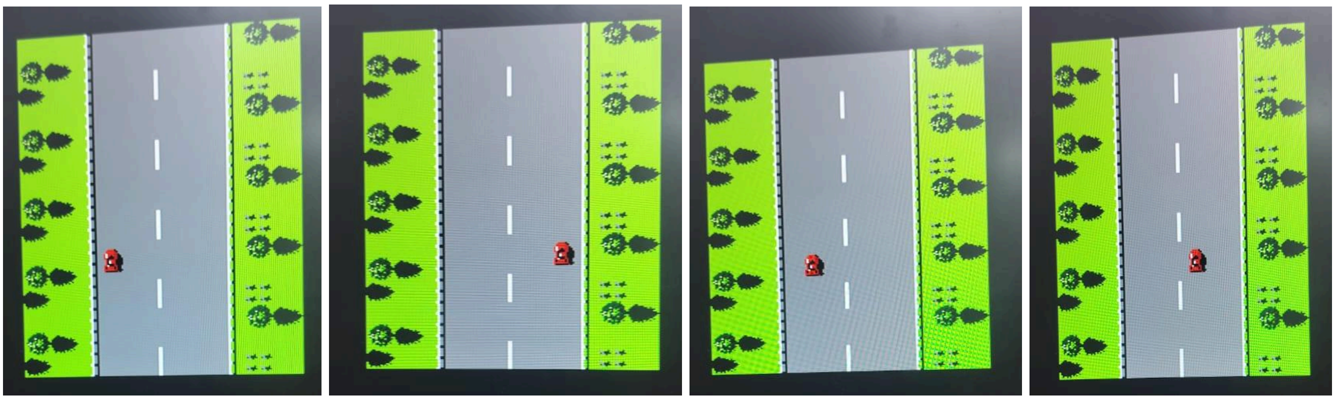
Tcl Console | Messages | Log | Reports | Design Runs

led Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy	Report Strategy
				144	**120**	**0**	**0**	**2**	**11/7/25, 1:58 PM**	**00:00:32**	**Vivado Synthesis Defaults (Vivado Synthesis 2022)**	**Vivado Synthesis Default Reports (Vivado**
0	49 CW, 3 Warn			193	130	14	0	2	11/7/25, 1:59 PM	00:00:47	Vivado Implementation Defaults (Vivado Implementation 2022)	Vivado Implementation Default Reports (Vivado I
				52	10	13.5	0	0	10/31/25, 2:17 PM	00:01:05	Vivado Synthesis Defaults (Vivado Synthesis 2022)	Vivado Synthesis Default Reports (Vivado Synthe
				0	0	0.5	0	0	10/31/25, 2:19 PM	00:00:41	Vivado Synthesis Defaults (Vivado Synthesis 2022)	Vivado Synthesis Default Reports (Vivado Synthe

The 14 **BRAMs** are primarily consumed by the two large single-port ROMs (bg_rom and main_car_rom) required for storing the background and car sprite image data. The low usage of **LUTs (193)** and **FFs (130)** demonstrates that the Car_control_FSM and its clock divider, which manage the entire game's control flow, are highly efficient, requiring minimal combinatorial and sequential logic. The design is easily contained within the available FPGA resources.

5. On-Board Implementation

This implementation successfully displays the car position and the road background on the VGA. The following snippets show the car at left collision position, right collision position, and at some arbitrary position while moving left or right.

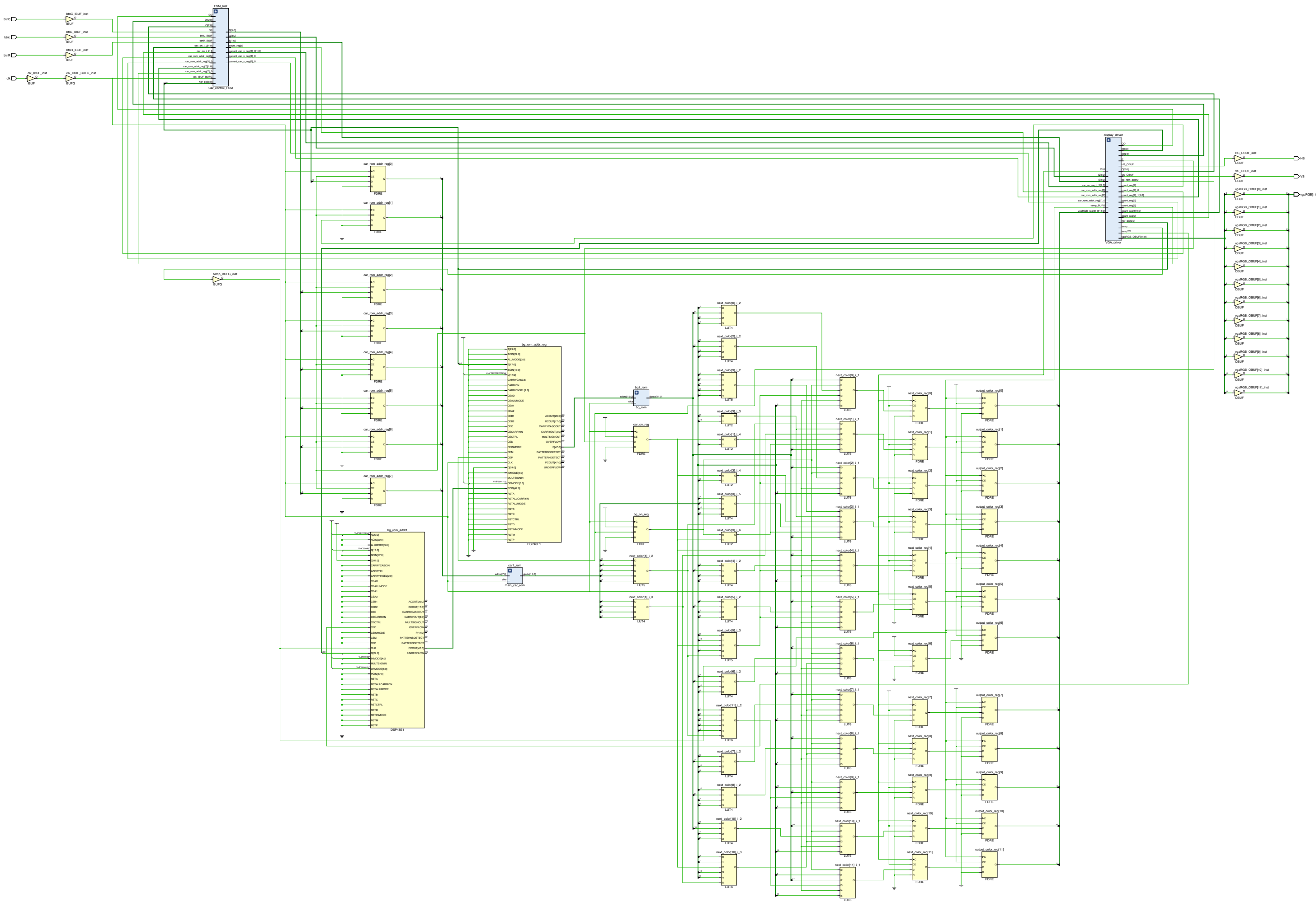


6. Conclusion

Part II successfully implemented the car's primary control mechanism using a 5-state Car_control_FSM. This module ensures **smooth horizontal movement** (at 10 Hz) and manages game-state transitions based on user input. The FSM correctly implemented the **look-ahead collision logic** to safely halt the car at the boundaries (244 and 304), transitioning the game into the COLLIDE state upon impact, where it awaits a restart via btnC. This fully functional control layer provides the interactive element required for the foundation of the *Road Fighter* game.

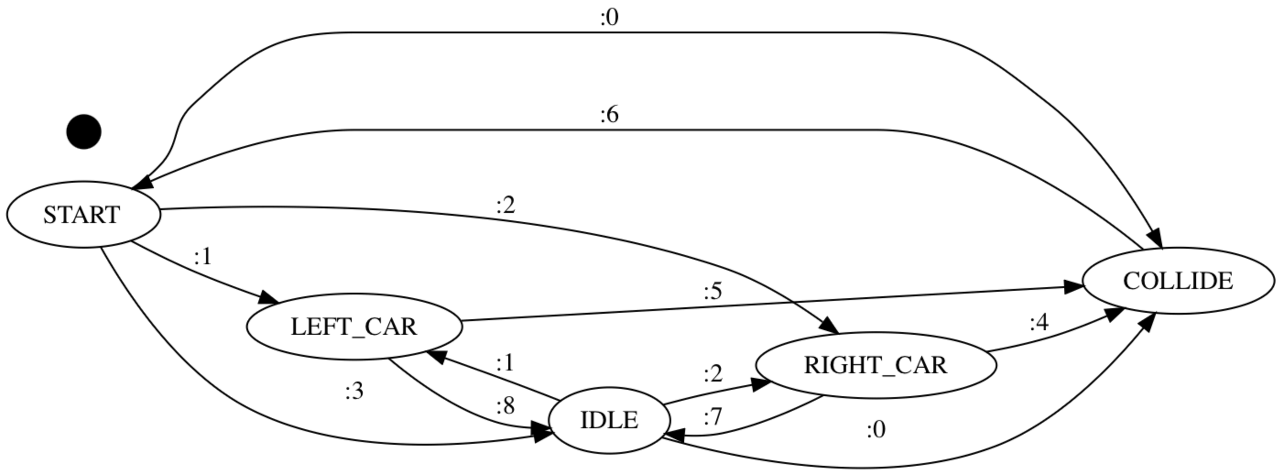
7. Generated Schematic

The Generated Schematic for part2 of this project is attached below



btnC
btnL
btnR
clk
btnC_IBUF_inst
btnL_IBUF_inst
btnR_IBUF_inst
clk_IBUF_inst
clk_IBUF_BUFG_inst
IBUF
BUFG
FSM_Inst
Car_control_FSM

8. FSM STATE DIAGRAMS



This is the state diagram of the FSM used in our code (default next_state is not shown

Input singals on which states changes

0 = (at_left_now || at_right_now)
1 = btnL
2 = btnR
3 = none
4 = (at_right_now || will_hit_right)
5 = (at_left_now || will_hit_left)
6 = btnC
7 = !btnR
8 = !btnL