COL215 – Digital Logic and System Design
Department of Computer Science & Engineering, IIT Delhi
Semester I, 2025-26
Lab Assignment 8

Road fighter game

1 Introduction

The objective of the assignment is: **Implementing the car racing arcade game on a basys 3 board**. The assignment is intended to result in below learning outcomes:

- Interfacing VGA with basys 3 board.
- Rendering smooth animation of still images on VGA.
- Implementing a **'Finite State Machine (FSM)'** to control different aspects of the game.

Diagrams in this document are only for the purpose of illustration and explaining the problem statement, and don't represent the exact design to be implemented. Design choices can vary among groups and must be carefully explained in the assignment report.

2 Problem Description

The assignment is divided into multiple parts with separate deadlines. Parts I and II are mandatory. Part III is optional; bonus marks will be awarded if you complete this part.

2.1 Part I: Displaying an image using VGA port

The Video Graphics Array (VGA) connector has 15 pins: three lines for Red-Blue-Green (RGB), two for synchronisation (sync) signals, and the rest are ground pins to regulate the current. Refer to the Basys3 Reference Manual, section 7.1, for pin configuration details.

The Basys 3 board has a VGA port to output an image to a monitor. A display controller module is needed to drive an externally connected VGA display. We have provided the `VGA_driver.v` module to configure the VGA port for 640x480 resolution. It generates the necessary HSYNC and VSYNC singals required for the VGA operations. The module has three sub-modules: (i) Pixel clock generator, (ii) Horizontal pixel counter, and (iii) Vertical pixel counter. For full details on the VGA timings and module design, refer to **COL215_VGA_timing_document**.

Along with the display controller, we have provided an initial template code (`Display_sprite.v`) which uses the VGA controller to display a still image. The still image consists of the background image from the road fighter game and the red car image (called a *sprite* in graphics terminology).

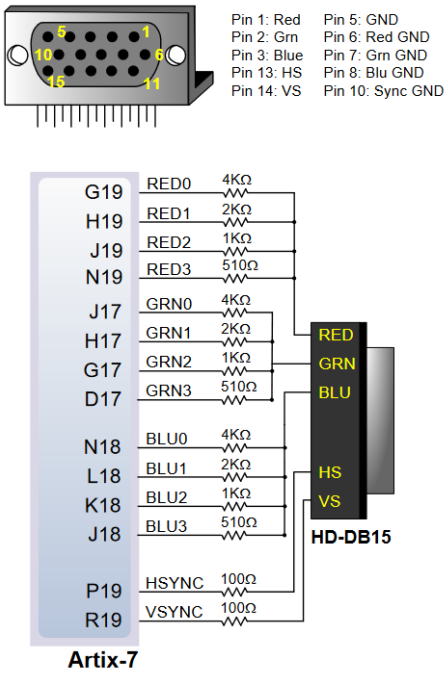
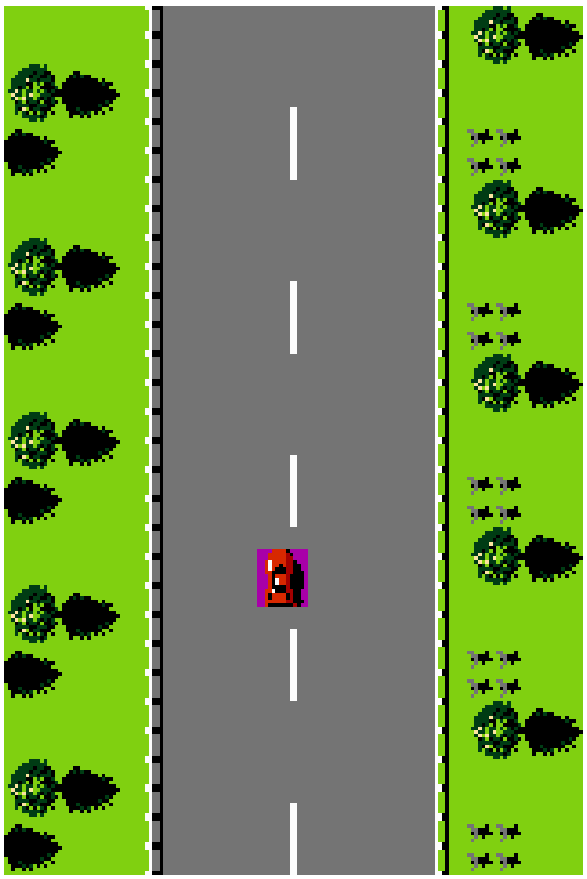


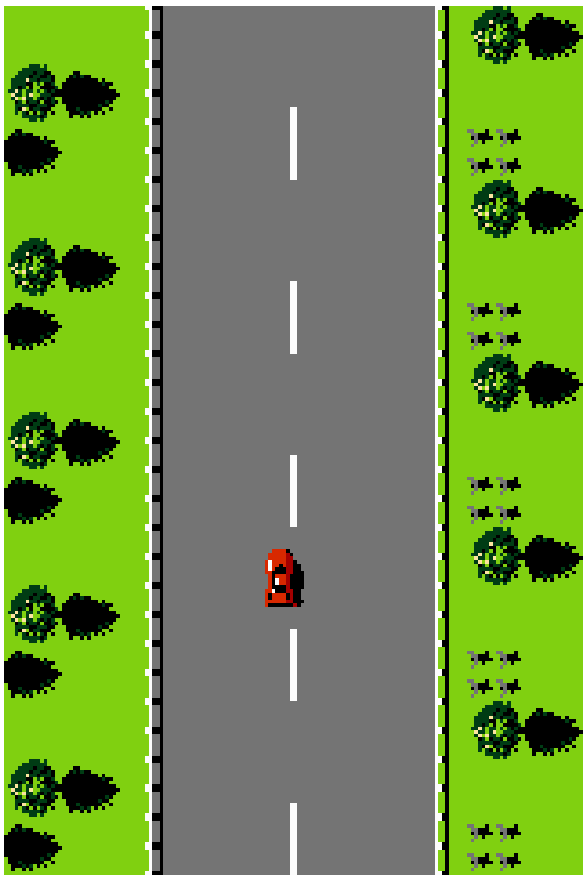
Figure 1: Pin connector for VGA basys 3 board

Your first task is to:

1. Create a single port ROM with the name `bg_rom`, with 12 bits width and 38400 as depth of port A. Initialize it with `bg1_testing.coe` COE file.
2. Create a single port ROM with the name `main_car_rom`, with 12 bits width and 224 as depth of port A. Initialize it with `main_car_testing.coe` COE file.
3. Create a Verilog source file. Copy the code provided in `Display_sprite.v`.
4. Write a test bench to run a simulation and analyze the generated output signals such as HS and VS. **Report the gap between successive HS and VS pulses (in number of clock cycles).**
5. Perform Synthesis and Implementation after updating the xdc file appropriately. Load the generated bitstream onto the FPGA and check the displayed image on a VGA monitor. It should match the image shown in Figure 2a.
6. Refine the code to remove the pink color from the red car in Figure 2a.
   - The pink color acts as a way to denote a transparent background in the graphics sprites.
   - Simple logic to implement transparency is to check the output value of `main_car_rom` and pass it to the VGA output port if it is not equal to pink (The 12-bit value of pink color is `12'b101000001010`); otherwise pass the output value of `bg_rom` to the VGA port.
   - You should modify the *always* block `MUX_VGA_OUTPUT` given in the template code. The final image should look like Figure 2b.



(a) Original Image with pink color not removed



(b) Image with pink color removed

2.2 Part II: Control logic to move car and detect collision

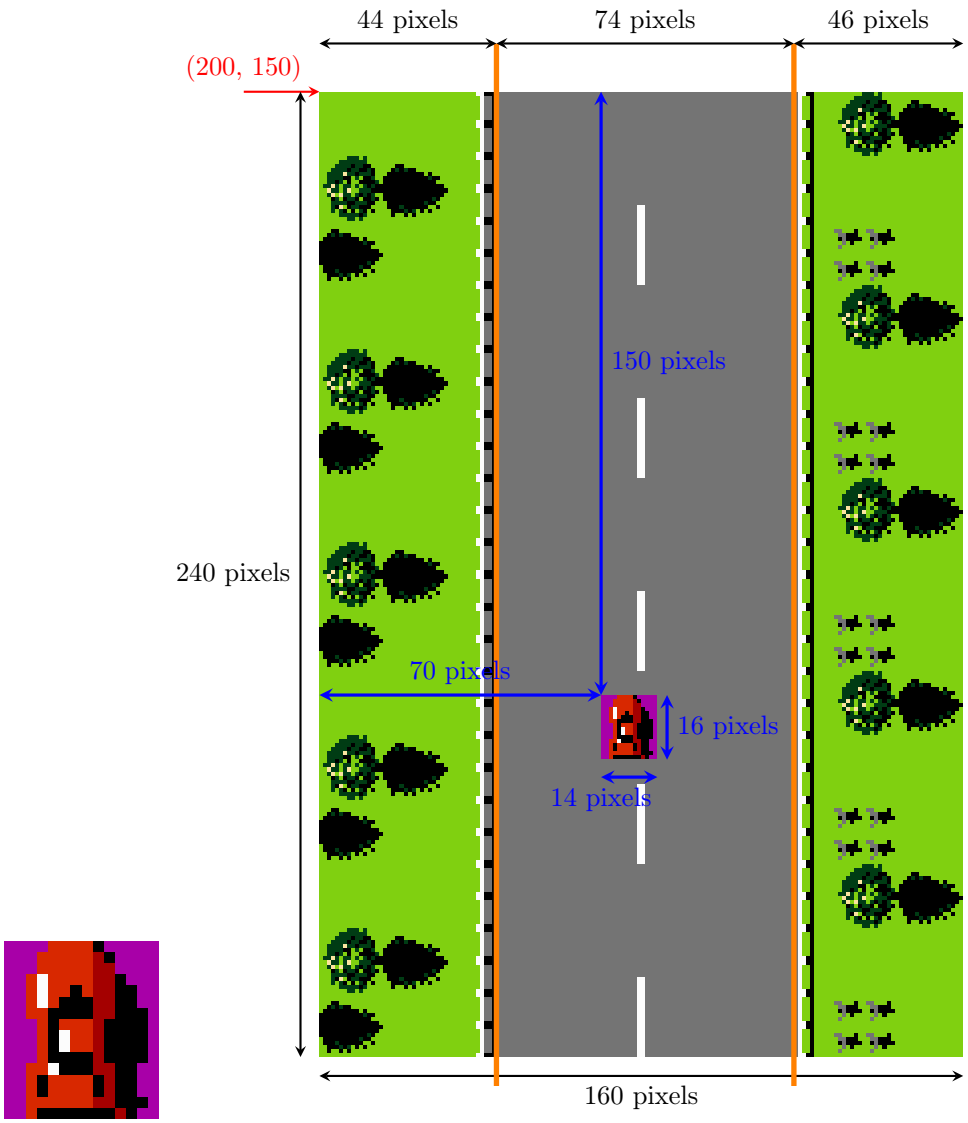
The second part of the assignment has two control logic to implement:

1. To **move the car horizontally**, right or left, using the two push buttons on the basys 3 board (**BTNL and BTNR**).
2. Use push button **BTNC** to restart the game.
3. To detect car **collision** with road boundaries.

When the right/left push button is pressed, the car should move in the right/left direction until the push button is released or the car collides with either end of the road. You will need a **Finite State Machine (FSM)** to update the direction of the car on the screen and detect the collision with either side of the road.

2.2.1 Collision logic

The main car image is a rectangle of size `(14x16 pixels)`, together with pink background, red car, and shadow. The rectangular boundary is the hit box of the car. The hit box dictates the area (number of pixels) the car takes on the frame; anything inside the box is a car and outside is the background image (road). This is irrespective of the actual size of the car, which is only used for a proper visual representation. **Using a rectangular box makes pixel manipulation simpler and faster to implement in the hardware**. The main car graphic is shown in Figure 3a, with the pink boundary as the hit box.



(a) Car image (b) Pixel width and distance for car and background image with hit box

Figure 3

The background image is drawn from `(200, 150)` and car at `(270, 300)` as shown in Figure 3b. The background origin offset can be changed via the display_sprite parameters and the car origin can be changed using the `car_x` and `car_y`. In the initial state, the car is drawn at fixed coordinates of `(270, 300)` which means the top left corner of the car box is drawn at `(270,300)` and the rest of the image has co-ordinates relative to this corner. To move the car right, the top left corner should be drawn at `(272,300)`. Similarly, to move it left, the corner should be drawn at `(268, 300)`. Figures 4 and 5 show the updated position of the car on the right and left, respectively. The figures show both images: car with hit box and car without hit box.

Following are the design decisions left to you:

- The number of pixels by which car is moved when push button is pressed should be decided based on smoothness of motion on the display.
- Keeping the push button pressed should move car continuously in the corresponding direction. The push button is to be pressed again for further movement.

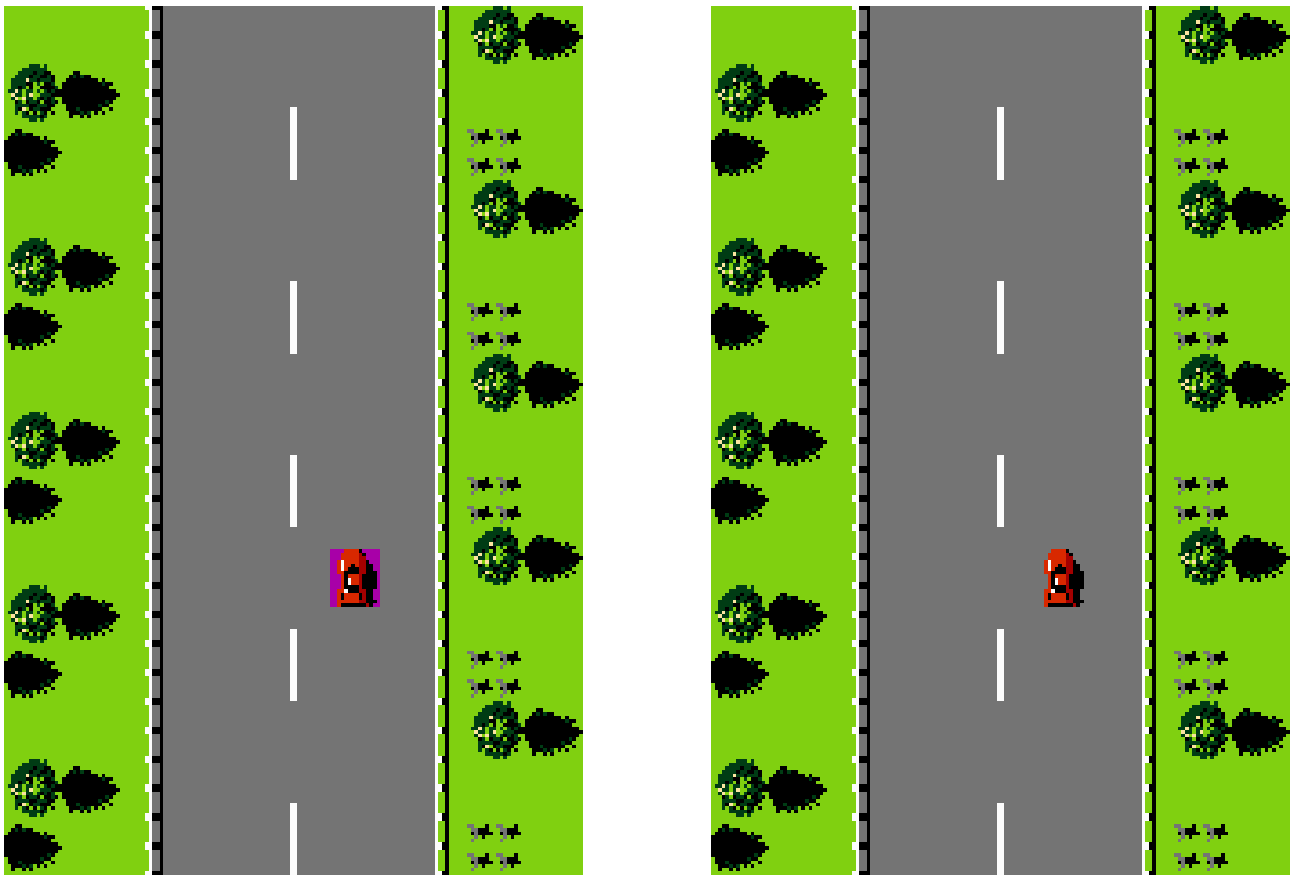


Figure 4: Car image when moved in right direction

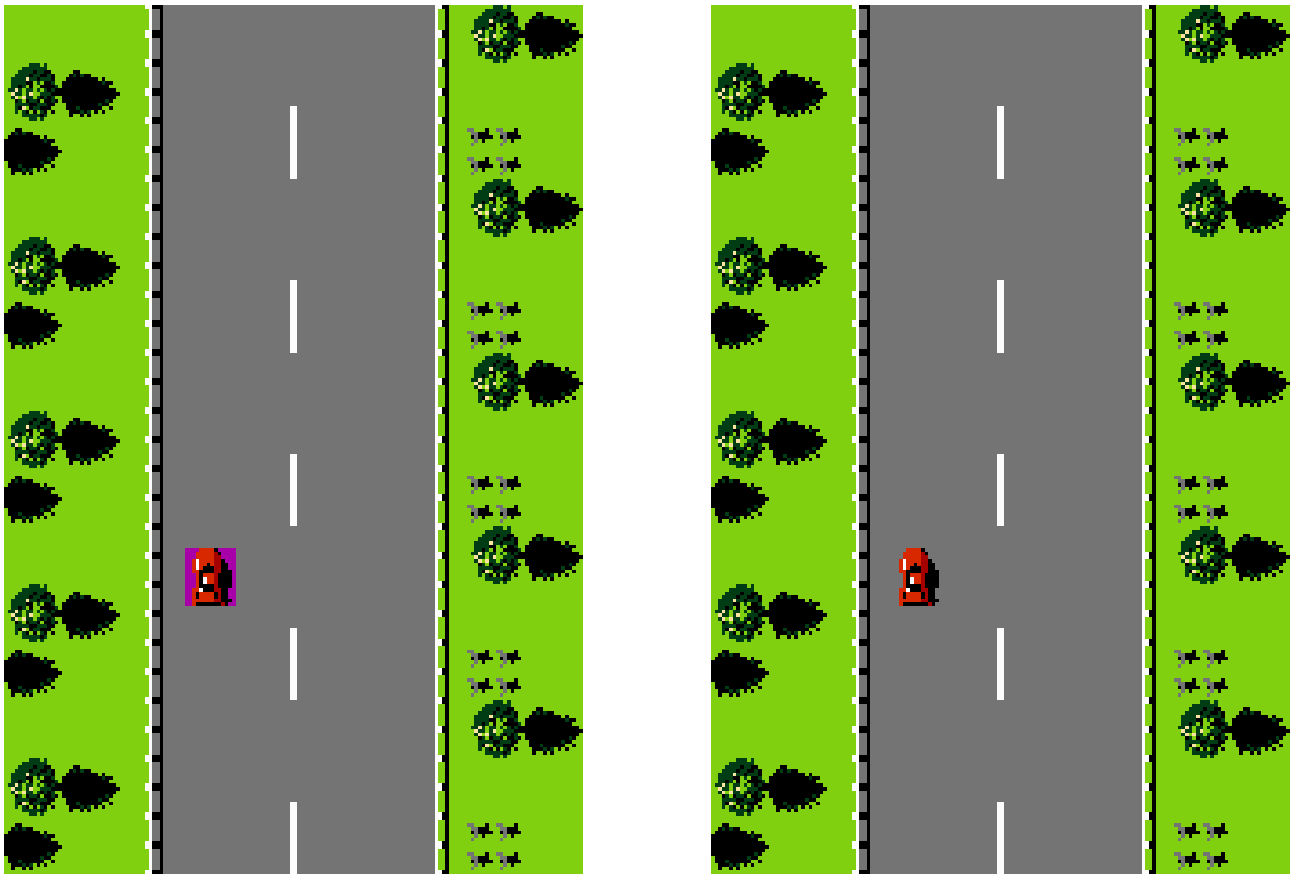


Figure 5: Car image when moved in left direction

As shown in Figure 3b via orange vertical lines, the left road boundary starts at `x=44` and the right boundary at `x=118`. So, a collision will occur whenever the horizontal coordinate of the car is less than 44 or greater than 118 relative to the background image, as shown in Figure 6. With respect to monitor resolution `(640x480)`, the boundary pixel value will be `x=200+44=244` and `x=200+118=318`, so a collision will occur when:

$$car\_top\_left\_x < 244$$
$$car\_top\_right\_x > 318 \textbf{ or } car\_top\_left\_x + 14 > 318$$

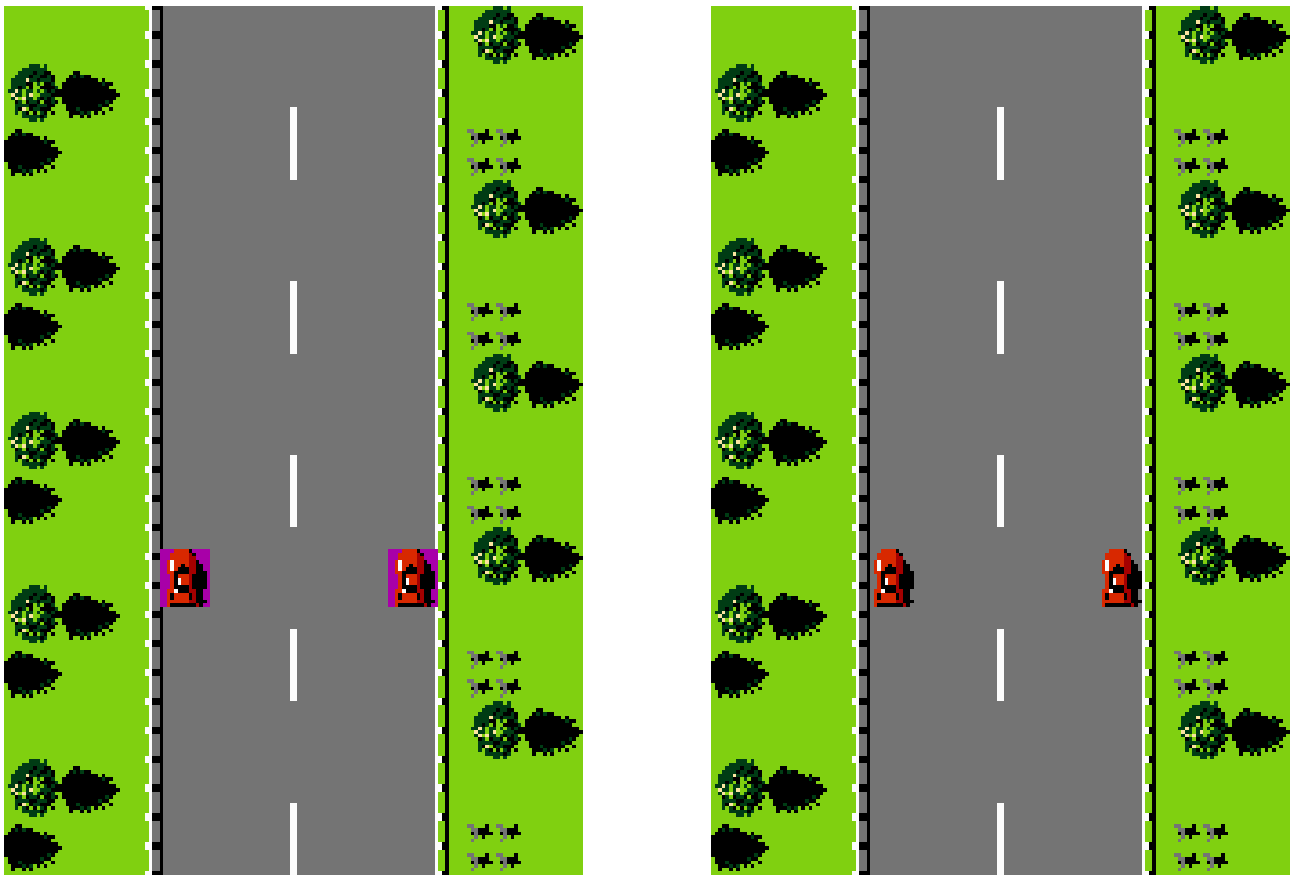


Figure 6: Car collision boundary

2.2.2 States in control FSM

1. The initial state (**START**): the game starts with the main red car at fixed coordinates of (270, 300).
2. **RIGHT_CAR**: The main car is horizontally moving in the right direction.
3. **LEFT_CAR**: The main car is horizontally moving in the left direction.
4. **COLLIDE**: The main car crosses either side of the road. Game is over; to re-start the game press **BTNC**.
5. **IDLE**: When no push button is pressed and the car is within the length of the road.

The state diagram is shown in Figure 7 showing the main state transitions (fill in any missing transitions). Your task is to implement the FSM in Verilog using the template given in Section 2.2.3.**You can modify the given FSM (add or remove the states/transitions) based on your final design.** You are allowed to reuse the push button modules from earlier assignments.

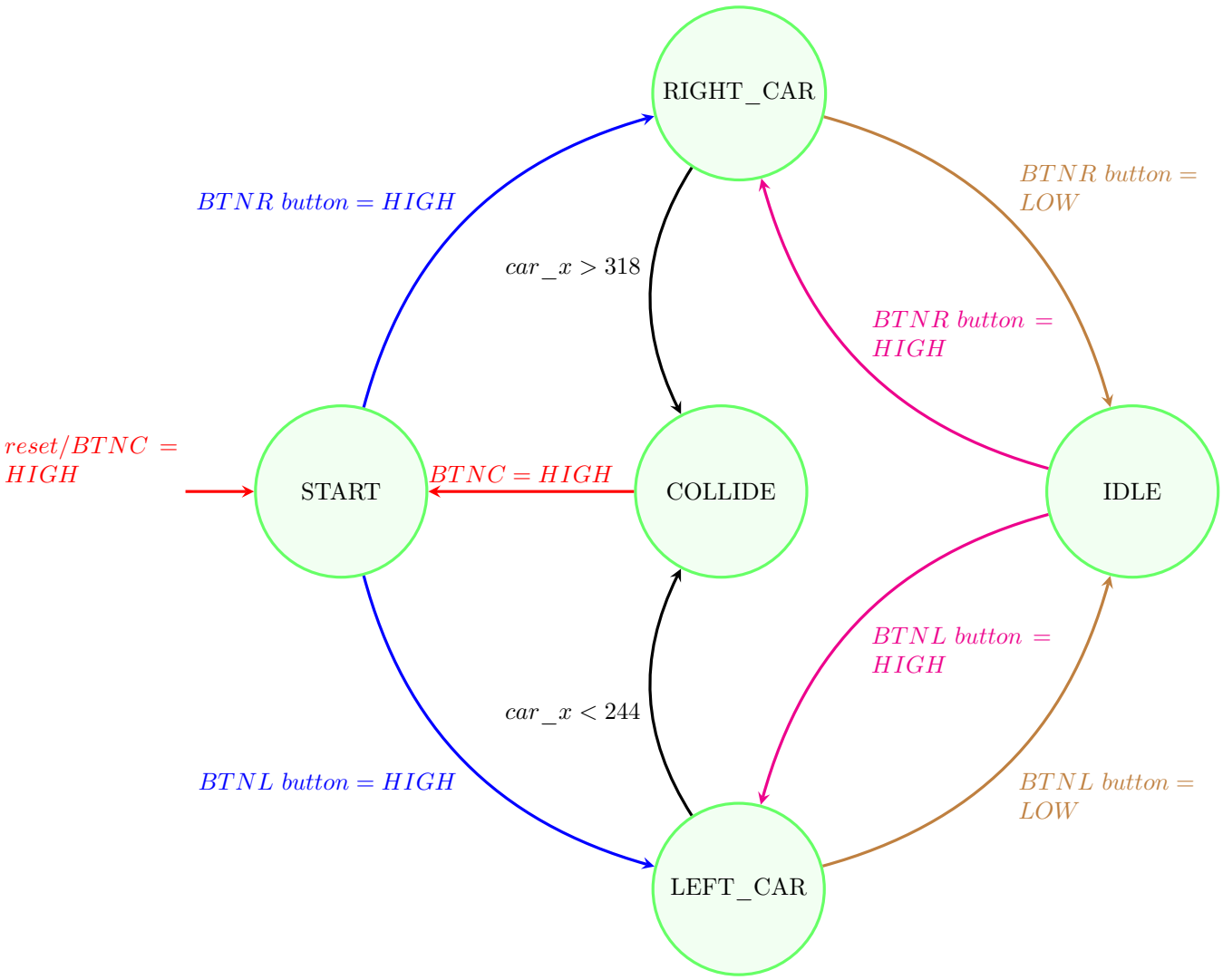


Figure 7: FSM for car movement and collision

2.2.3 Creating Finite State Machines in Verilog

In this section, we illustrate how to specify an FSM in Verilog. An example FSM shown in Figure 8. Its job is to control the operation of a programmable adder/subtractor by sending a signal that instructs the component to ADD or SUBTRACT.

The FSM consists of 2 states, ADD and SUB, along with input flags M (to change the operation) and DONE (to know when the current operation is completed); and output flag `cntrl_add_sub` to indicate the operation in the programmable adder/subtractor.

The FSM, specified in Verilog, consists of two parts:

1. **State Register**: In this block, the state is updated on the rising clock edge or initialized to a default state if the reset flag is set HIGH.

2. **Combinational Block**: This consists of the logic to update the next state based on the flag values and the current state.

The following is the Verilog template for implementing an FSM. You can see that the state of the system is captured in the State Register and updated every positive clock edge, and the work performed in one clock cycle is indicated/controlled in the Combinational block.

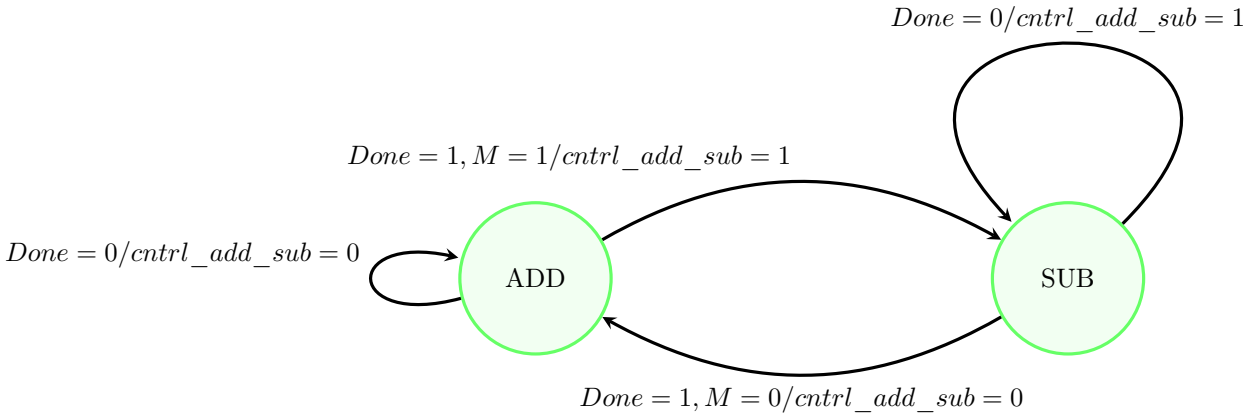


Figure 8: FSM for adder/subtractor

```
module add_sub_fsm (
    input  wire clk,
    input  wire reset,
    input  wire Done,
    input  wire M,
    output reg  cntrl_add_sub
);

    // State encoding
    localparam ADD = 0;
    localparam SUB = 1;

    reg current_state, next_state;

    // Next-state and output logic (combinational block)
    always @ (*) begin
        next_state = current_state;

        case (current_state)
            ADD: begin
                if (Done && M) begin
                    next_state = SUB;
                    cntrl_add_sub = 1'b1;  // SUB
                end
                else if (!Done) begin
                    next_state = ADD;
                    cntrl_add_sub = 1'b0;  // ADD
                end
            end

            SUB: begin
                if (Done && !M) begin
                    next_state = ADD;
                    cntrl_add_sub = 1'b0;  // ADD
                end
                else if (!Done) begin
                    next_state = SUB;
                    cntrl_add_sub = 1'b1;  // SUB
                end
            end

            default: begin
                next_state = ADD;
                cntrl_add_sub = 1'b0;
```

```
            end
        endcase
    end

    // State register (sequential block)
    always @ (posedge clk or posedge reset) begin
        if (reset)
            current_state <= ADD;
        else
            current_state <= next_state;
    end

endmodule
```

2.3 Part III: Displaying a rival car

The task is to generate a rival car (different color) in a random position at the top of the road and move it vertically downward with constant speed. The flow is depicted in Figure 9. The rival car starts at the top of the road and vertically moves down the image. If no collision occurs with main car, then the rival car is re-generated again from the top.

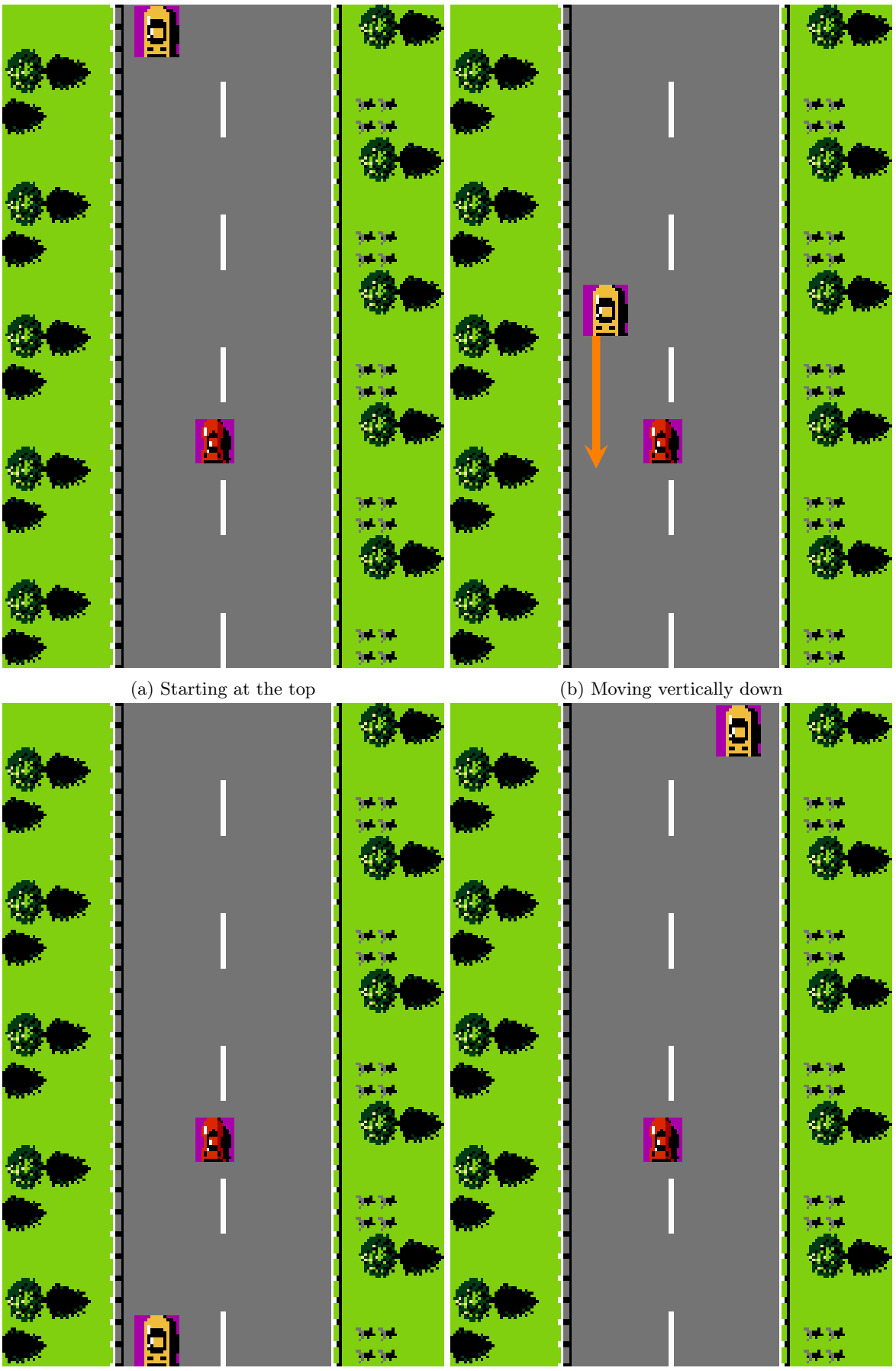
You are provided `rival_car.coe`. Implement the following tasks.

1. Implement a 8-bit pseudo random generator. Details are described in Section 2.3.1.
2. Create a single port ROM with the name `rival_car_rom`, with 12 bits width and 224 as depth of port A. Initialize it with `Rival_car_testing.coe` COE file.
3. Implement logic to randomly get horizontal pixel between `44` and `104` pixel using pseudo random generator and vertical pixel value fixed at (`OFFSET_BG_Y`). Refer to Figure 9 indicating the rival car starting position at the top.
4. Implement logic to move the rival car vertically down until it reaches the bottom boundary of the road.
5. Augment the FSM implemented to detect collision between main and rival car. Collision occurs when the hit boxes of both cars overlap.

The vertical movement of the car can be done by updating the top left corner coordinate after end of a video frame. A video frame ends when (`hor_pix` = 799) and (`ver_pix` = 479). Rival car speed will be very fast if pixel values are updated after every frame-end; instead the value should be changed after 10-15 frames.

Approximate logic to achieve vertical movement of rival car:

1. Maintain rival car position in a register `(rival_x, rival_y)` and frame count in `frame_reg_count`
2. When (`hor_pix` = 799) and (`ver_pix` = 479), update the `frame_reg_count`.
3. Fixed value for the maximum count of frame and by what amount y-coordinate is increased. **These two are design decisions**. Generally, frame count should be in range (10-60).
4. When `frame_reg_count` reaches a threshold, update the `rival_y` and reset frame counter register.



(a) Starting at the top

(b) Moving vertically down

(c) No collision with red car and it reaches bottom

(d) Restarting from top at new location

Figure 9: Rival car movement flow from top to down

2.3.1 8-bit Pseudo Random Number Generator

To generate a random number, the Linear Feedback Shift Register (LFSR) can be used. The LFSR is a hardware synthesizable approach for generating a random number in a fixed sequence. It is designed using a series of D-flipflops, wherein a linear function, e.g., XOR of some selected flop outputs, is used as the feedback.

The block diagram for one such 8-bit LFSR is shown in Figure 10. In order to achieve sufficient randomness, the LFSR should have a reset value as a parameter, sufficiently random. In this assignment, the parameter value for a group should be the bitwise XOR of 8 LSBs of two kerberos IDs. For groups having a single member, use 8 LSBs of your kerberos ID.

Figure 11 demonstrates the states of an LFSR and how the value gets updated across the clock cycles based on shifting of the bits.

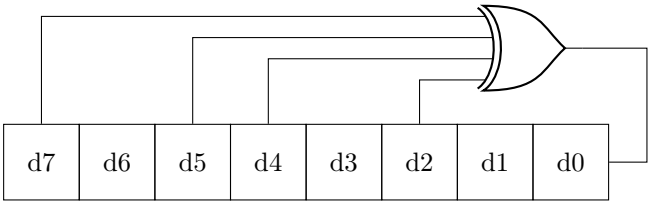
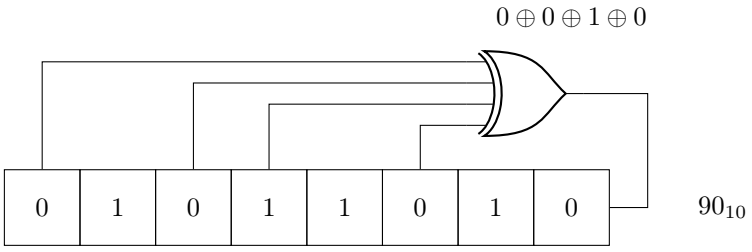
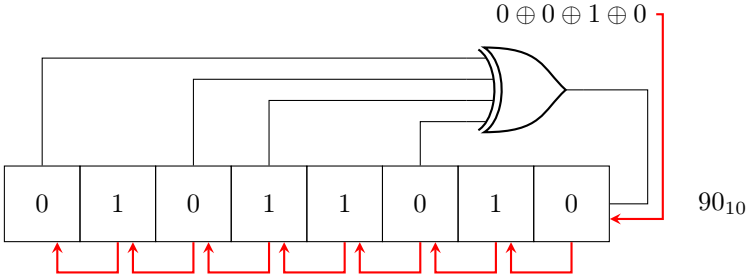


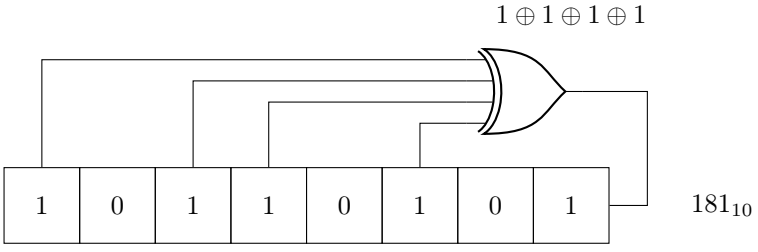
Figure 10: 8-bit LFSR



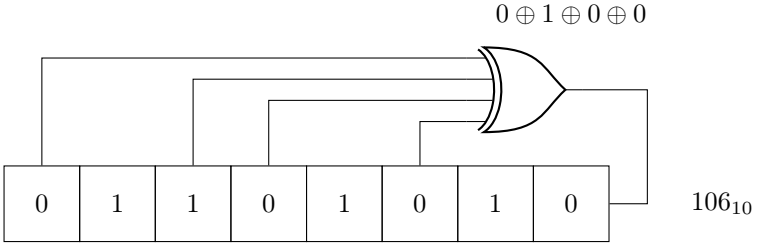
(a) Clock cycle I



(b) Shifting of the bits



(c) Clock cycle II



(d) Clock cycle III

Figure 11: States of LFSR

3 Submission and Demo Instructions

1. Demo should be given in the assigned lab slot itself.
2. You are required to submit the following on Gradescope.
    - Verilog files for all the designed modules. Only the modules written by you are needed. The IPs generated, i.e., distributed memories must not be submitted.
    - Verilog files for the testbench and the simulation of all the operations.
    - Warning-clean constraint file (.xdc), bit file.
    - Questionnaire corresponding to the assignment.
    - A short report (5-6 pages) outlining simulation snapshots, utilization data (BRAMs, DSPs, LUTs, and FFs), and generated schematics. The FSM diagram(s) are mandatory. The input and output signal names should be the same as ones used in Verilog files. Explain your design decisions. Snippets in the report without explanation will result in penalty.

NOTE:

- Ensure all the files in the zip folder are correctly visible on gradescope. Later requests about technical glitches with gradescope and files being in zip folder and not visible on gradescope and being part of report wouldn't be considered anymore.
- The report must be in pdf format only (docx will not be considered any more) and should consist of the points mentioned in the Submission section. Not mentioning utilization data specifically in the report (and relying on synthesis screenshots) will not be considered anymore.
- The submission must have only the above listed items. Submitting additional files such as video of board functionality, separate utilization report, vivado xpr or dcp files, etc. will result in penalty.

We advise you to be ready with your design before the lab session, and during the session, perform validation by downloading it into the FPGA board.

4 Resources references

- IEEE document: `https://ieeexplore.ieee.org/document/1620780`
- Basys 3 board reference manual: `https://digilent.com/reference/_media/basys3:basys3_rm.pdf`
- Online Verilog simulator: `https://www.edaplayground.com`