CSE 100/L

Lab 6: Subway Slugging

December 3rd, 2025

Ceferino Sacramento

Lab Section #1

9:50 - 11:40 AM

1. Description

Subway Slugging, a game designed for the Basys3 Board and to be connected to a monitor via VGA. The game works by using the buttons on the board to start, and to move the player left, right, or hover to avoid the randomly sized, randomly timed trains coming down the screen. If you avoid a train and it passes by your player, your score increments. If you don't avoid the train and you crash, the game will freeze, and your player will flash until you reset the game.

2. Design

2.1 PixelAddress

Description:

PixelAddress is used to traverse through the pixels on the monitor screen to be able to display what we want. It traverses through the columns first, then the next row. The active region is 640 x 480 pixels, and the whole region is 800 x 525. Hsync and Vsync are signals for synchronization used to signal the display to start a new frame.

Logic: I implemented PixelAddress by using two 16-bit counters; one counter is used for the horizontal, which counts through all the columns. Once it counts through 799 times, representing that it has gone through the 800 pixels, wide the display is, it will then count vertically to represent going onto the next row. I assigned the active region to be from 640 x 48,0, and it is active low, so it is the opposite of what would be expected if it were active high. Then assigning HSync and Vsync to when the counter hits 655 to 750, and 489 to 490, also keeping in mind that it is active low.

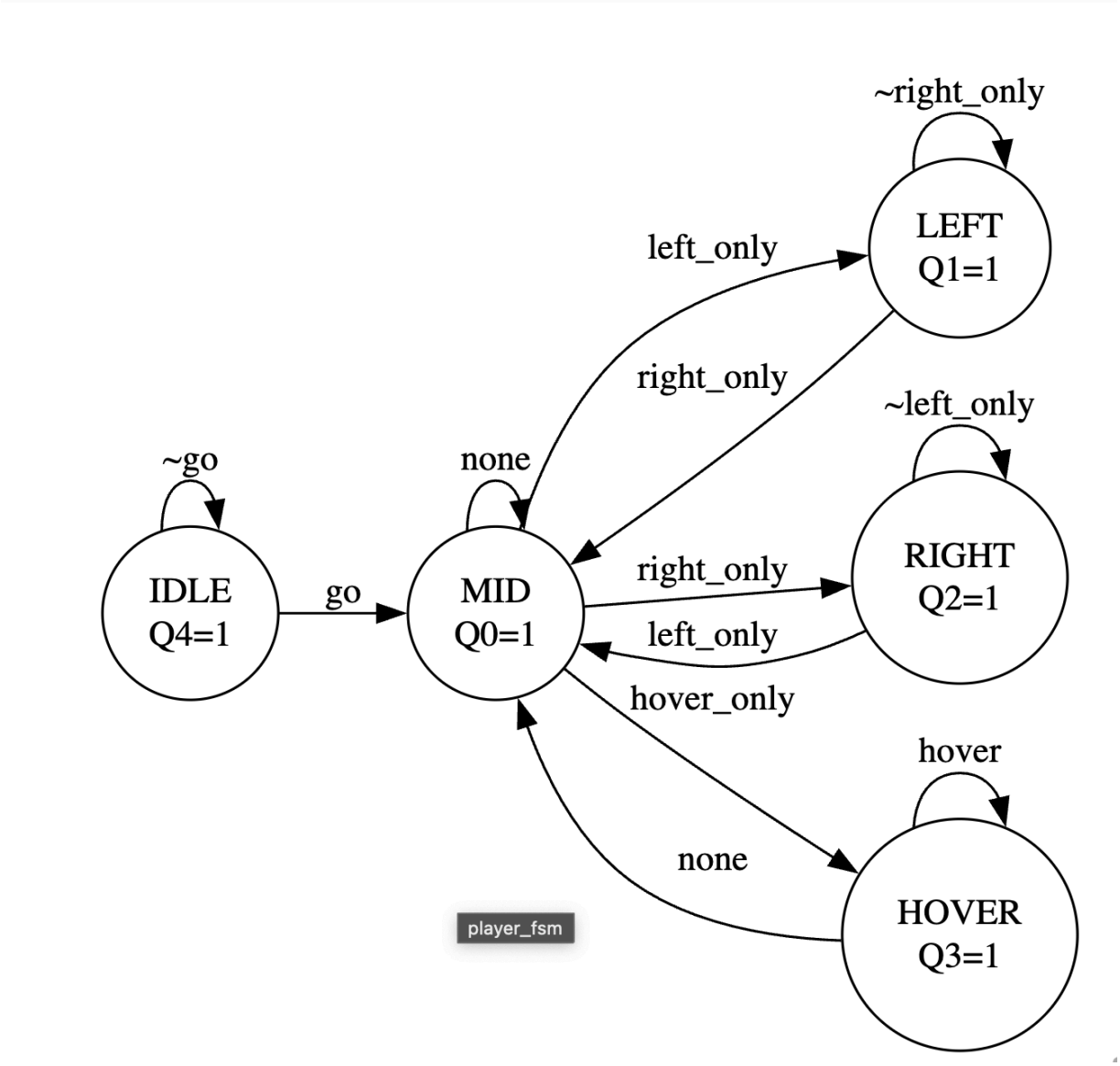
Code for PixelAddress:

```
module pixeladdress(
input clk_i,
output [15:0] hCount,
output [15:0] vCount,
output active_region,
output Hsync,
output Vsync
    );
wire hWrap  = (hCount == 16'd799);
countUD16L h (.clk_i(clk_i), .up_i(~hWrap), .dw_i(1'b0), .ld_i(hWrap), .Din_i(16'd0), .Q_o(hCount
wire vMax = (vCount == 16'd524);
wire vEnd  = hWrap & vMax;
wire vWrap = hWrap & ~vMax;
countUD16L v (.clk_i(clk_i), .up_i(vWrap), .dw_i(1'b0), .ld_i(vEnd), .Din_i(16'b0), .Q_o(vCount))
assign active_region = (hCount >= 16'd640) | (vCount >= 16'd480); // may need change
assign Hsync = (hCount < 16'd655) | (hCount > 16'd750); // may need change
assign Vsync = (vCount < 16'd489) | (vCount > 16'd490); // may need change
endmodule
```

2.2 Player State Machine

Description: State Machine for the player, which uses one-hot encoding so that the player can start the game, move left, right, or hover depending on the inputs.

State Machine Diagram:



Logic Equations:

State	1-Hot Encoding	Logic
Idle	10000 (4)	Idle = (Q[4] & ~go)
Middle	01000 (3)	Middle = Q[4] & go | Q[0] & none | Q[1] & right_only | Q[2] & left_only | Q[3] & none;

Left	00100 (2)	Left = ((Q[0] & left_only) |(Q[1] & ~right_only));
Right	00010 (1)	Right = ((Q[0] & right_only) | (Q[2] & ~left_only));
Hover	00001 (0)	Hover = (Q[0] & hover_only)| Q[3] & hover;

Table 1

Code for PlayerFSM:

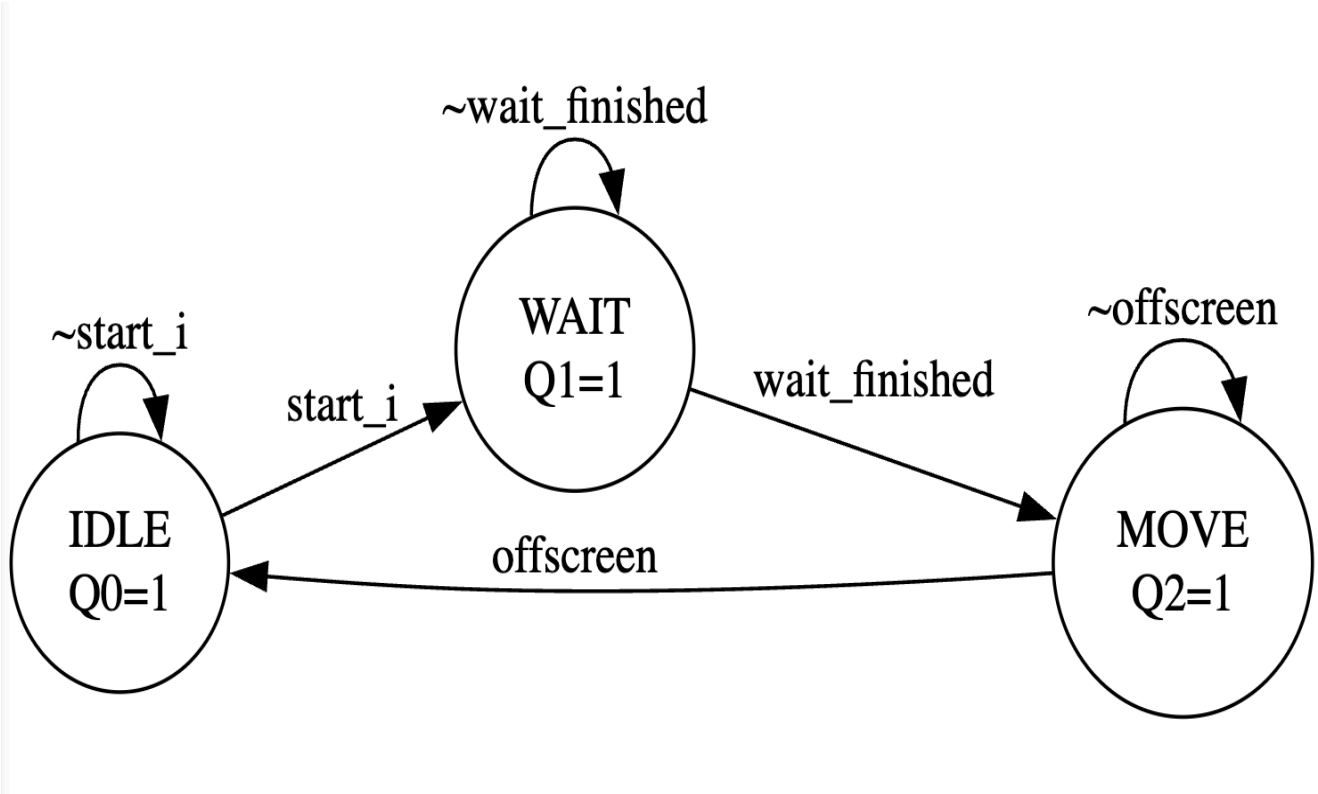
```
module player_Fsm(
input clk_i,
input left,
input Go,
input right,
input hover,
output [1:0] lane,
output hover_on
    );
    wire left_only = left & ~right & ~hover;
    wire right_only = right & ~left & ~hover;
    wire none = ~right & ~left & ~hover;
    wire hover_only = ~right & ~left & hover;
    wire [4:0] D;
    wire [4:0] Q;
    wire go = Go & none;
    //idle
    assign D[4] = (Q[4] & ~go);
    //aka middle
    assign D[0] = Q[4] & go | Q[0] & none | Q[1] & right_only | Q[2] & left_only | Q[3] & none;
    //left
    assign D[1] = ((Q[0] & left_only)  |(Q[1] & ~right_only));
    // right
    assign D[2] = ((Q[0] & right_only) | (Q[2] & ~left_only));
    //hover
    assign D[3] = (Q[0] & hover_only)| Q[3] & hover;

    FDRE #(.INIT(1'b0)) Q0 (.C(clk_i), .R(1'b0), .CE(1'b1), .D(D[0]), .Q(Q[0]));
    FDRE #(.INIT(1'b0)) Q1 (.C(clk_i), .R(1'b0), .CE(1'b1), .D(D[1]), .Q(Q[1]));
    FDRE #(.INIT(1'b0)) Q2 (.C(clk_i), .R(1'b0), .CE(1'b1), .D(D[2]), .Q(Q[2]));
    FDRE #(.INIT(1'b0)) Q3 (.C(clk_i), .R(1'b0), .CE(1'b1), .D(D[3]), .Q(Q[3]));
    FDRE #(.INIT(1'b1)) Q4 (.C(clk_i), .R(1'b0), .CE(1'b1), .D(D[4]), .Q(Q[4]));
    assign hover_on = Q[3];
    assign lane = Q[1] ? 2'd0 : (Q[2] ? 2'd2 : 2'd1);
endmodule
```

2.3 Train State Machine

Description: The train state machine was implemented in a way so that the random trains know when to start, wait, and move. I assigned random times and lengths in this same file using my LFSR to generate random lengths of 60 + 0-63, and random times of 0-127, using an 8-bit LFSR module that I had

previously built before.

State Machine Diagram:



Logic Equations:

State	1-Hot Encoding	Logic
Idle	100(3)	Idle = (Q[0] & ~start_i) | (Q[2] & offscreen);
WAIT	010 (2)	WAIT = (Q[0] & start_i) | (Q[1] & ~wait_finished);
MOVE	001(1)	MOVE = (Q[1] & wait_finished) | (Q[2] & ~offscreen);

Table 2

Code For TrainFSM:

```
module fsmtrain(
    input clk_i,
    input frame_clocked,
    input start_i,
    output [15:0] train_top,
    output [15:0] train_bottom,
    output running
    );
    wire [2:0] D;
    wire [2:0] Q;
//  wire [6:0] rnd;
    wire [15:0] screen_height = 16'd480;
    wire [15:0] train_start_y = 16'd0;
    wire [15:0] bottom_q;
    wire [15:0] bottom_d;
    wire [15:0] bottom_inc1 = bottom_q + 16'd1;

    // length register
    wire [15:0] length_q;
    wire [15:0] length_d;

    // top = bottom - length
    wire [15:0] top_raw = bottom_q - length_q;
    wire [15:0] actual_top = (bottom_q > length_q) ? top_raw : 16'd0;

    wire offscreen = (actual_top >= screen_height);
    wire [6:0] rnd;
    lfsr rng (.clk_i(clk_i), .frame_clocked(frame_clocked), .rand (rnd));

    // random wait time
    wire [7:0] rand_wait_raw = {1'b0, rnd};
    wire [7:0] rand_wait = (rand_wait_raw == 8'd0) ? 8'd1 : rand_wait_raw;
//    wire [7:0] rand_wait = rand_wait_raw + 8'd32;

    wire [5:0] rand_set = rnd[5:0];
    wire [15:0] rand_length = 16'd60 + {10'd0, rand_set};  // 60 to 123

    wire [7:0] wait_q;
    wire wait_finished;
    time_counter wait_counter (
    .clk_i(clk_i),
    .ld_i(Q[0] & start_i), // load random time when leaving idle
    .dw_i(Q[1] & frame_clocked & ~wait_finished), // count down 1 per frame in wait
    .Din_i(rand_wait),
    .q_o(wait_q)
    );
    assign wait_finished = (wait_q == 8'd0);

    //Idle
    assign D[0] = (Q[0] & ~start_i) | (Q[2] & offscreen);
    // Wait
    assign D[1] = (Q[0] & start_i) | (Q[1] & ~wait_finished);
    // Move
    assign D[2] = (Q[1] & wait_finished) | (Q[2] & ~offscreen);
    //one hot
    FDRE #(.INIT(1'b1)) idle (.C(clk_i), .R(1'b0), .CE(1'b1), .D(D[0]), .Q(Q[0]));
    FDRE #(.INIT(1'b0)) wait_ff (.C(clk_i), .R(1'b0), .CE(1'b1), .D(D[1]), .Q(Q[1]));
    FDRE #(.INIT(1'b0)) move (.C(clk_i),.R(1'b0), .CE(1'b1), .D(D[2]), .Q(Q[2]));

    assign bottom_d = (Q[0] | Q[1]) ? train_start_y : (Q[2] & frame_clocked & ~offscreen)  ? bottom_inc1  : bottom_q;
    flipflop train_positon (
    .clk_i(clk_i),
    .CE(1'b1),
    .D(bottom_d),
    .Q(bottom_q)
    );

    // load new random length on start_i
    assign length_d = (Q[0] & start_i) ? rand_length : length_q;
    flipflop length (
    .clk_i(clk_i),
    .CE(1'b1),
    .D(length_d),
    .Q(length_q)
    );
     assign train_top = actual_top;
    assign train_bottom = bottom_q;
    assign running = Q[2];
endmodule
```

2.4 Display

Description: The Display module is responsible for drawing everything that appears on the VGA screen. Using the current pixel position hCount and vCount, it decides the color of each pixel and assigns the RGB outputs. It first generates a white 8-pixel border around the 640×480 active region, then draws the vertical energy bar on the left side, the three train tracks on the right, and the 16×16 player sprite in one of three lanes. The module is where the train FSMs are "made visible": each fsmtrain instance provides the top and bottom coordinates of a train, and the Display module turns those into blue train rectangles that move smoothly down their track as frames advance. The player's hover state is also handled here: when hover_on is asserted, the player changes color and flashes using a small counter, while the energy bar slowly drains; when hover is off, the bar refills. The module also computes collisions by checking when any train rectangle overlaps the player rectangle (and god mode is not active), and it flashes the player on collision and latches a collision flag. Finally, Display generates score_pulse whenever the top of a moving train crosses the row just above the player, so the top-level score counter can increment whenever a train successfully passes the player.

Logic: Several groups of flip-flops implement timing and animation effects. A 4-bit hover_count counter plus a hover_phase flip-flop (5 FFs total) create a blinking player when hovering. Another 4-bit coll_count counter and collision_phase flip-flop (again 5 FFs) are used to flash the player after a collision. Additional flip-flops store the energy bar position, the player transition position, and the latched collision flag. Altogether, 16 flip-flops are used each time to control on-screen movement and flashing effects (hover animation, collision animation, energy motion, and player sliding). The six FSM train modules represent two trains per lane, each provides train_top, train_bottom, and a running flag; I gated those with the track bands to form train_on signals and also use the train_top == score_row comparisons to generate one-cycle score_pulse signals. At the end, a priority chain selects the pixel color: border has the highest priority, then energy, then the (possibly blinking) player, then any trains, then the tracks, and finally the black background. This makes the Display module the central place where all pixel-level logic, animation, energy level, player movement and hovering, scoring, and collision detection are combined into the final image seen on the screen.

(Code is too long to show code for the Display file)

2.5 Top

Description:

The top module is the top-level integration that connects all subsystems into a complete game running on the Basys3 board and driving the VGA monitor. It first uses labVGA_clks to derive the main game clock and the digit-select signal for the seven-segment display, then pixeladdress generates the pixel counters, active-region flag, and the Hsync/Vsync signals. Each pushbutton is passed through an edge_detector so

that the rest of the design receives clean one-cycle control pulses; these pulses are then gated with status signals such as player_center, collision, and energy_empty_flag to form the left, right, hover, and Go inputs to the player_Fsm. The resulting lane selection and hover flag are fed into solid_color along with the pixel timing. A pair of flip-flops detects the end of each VGA frame and produces a frame_clocked pulse, which is used together with a game_running flip-flop and a 16-bit frame_count register to time the game: while the game is running, the frame counter increments once per frame, and simple comparisons on its value open each track at different moments by asserting the start_*_trains signals that ultimately drive the train FSMs inside the display module. The top level also connects the score_pulse output from solid_color into a turkeyCounter that holds an 8-bit score, then uses a ring counter, selector, and hex-to-7-segment decoder to multiplex that score onto the four seven-segment digits, blanking the unused positions. Finally, it routes the RGB outputs from solid_color to the VGA color pins, exposes god_mode on a switch and LED, and therefore acts as the glue that coordinates clocks, control pulses, game timing, and I/O so that all lower-level FSMs and pixel logic operate together as one coherent system

(Code too long to show code for top)

3. Test and Simulation

My approach to testing this lab was very step-by-step, building up the game visuals and behavior gradually instead of trying to do everything at once. I started by getting a simple, solid color to show up on the VGA monitor screen just to confirm that the timing and basic output were working. Once that was stable, I added the white borders and made sure they were exactly 8 pixels wide around the screen. After that, I drew all of the main elements in fixed positions: the tracks on the right side, the energy bar on the left, and the player box in the middle lane, so I could verify that all the coordinates lined up the way I expected. With the static layout working, I next made the player move instantly between the three lanes in response to button presses, just to confirm that the lane selection and collision area were correct before worrying about animation. Only after that was behaving did I add the smooth movement between lanes, where the player slides horizontally instead of teleporting, and I watched the motion frame by frame to make sure it stayed centered in each lane at the end of a move. I also used the on-board LEDs as simple debug indicators to show internal states of my state machines and counters while the game was running, which helped me confirm that the state transitions were happening when I expected. A big part of the testing was chasing down edge cases, like what happens if the player starts hovering while changing lanes, how the energy bar drains and refills during hover, and whether energy and player position stay consistent when trains are spawning in different lanes. When something looked wrong on the screen, like the player or energy bar updating in the wrong place, I used those LEDs and carefully stepped through the logic until the transitions behaved correctly in all lanes and under all hover and collision conditions.

4. Results

Overall, the lab was implemented by building the game up visually, one piece at a time, and then layering in the control logic to make it interactive. Starting from a working VGA output, I added the borders, static player, tracks, trains, and energy bar, and then introduced movement, collisions, scoring, and hover effects until the final system behaved like a playable game. A key part of getting any of this to work was understanding how the pixel addressing and synchronization signals drive the display. The horizontal and vertical counters step through every pixel on the screen in a fixed order, and the Hsync and Vsync pulses mark the end of each line and the end of each frame. Once I understood that every visible object is defined purely in terms of ranges of hCount and vCount, it became much easier to reason about where rectangles should appear and how to move them smoothly over time. The timing of Hsync/Vsync and the active video region was also crucial for generating a clean frame_clocked signal: by detecting a specific (hCount, vCount) location at the bottom-right corner, I could create a single tick per frame, which I then used to update train positions, player transitions, and the energy bar at a consistent rate. In the final result, all of these pieces, correct sync timing, pixel address decoding, and frame-based updates, come together so that the VGA monitor shows a stable image with smoothly moving trains and a player that responds predictably to inputs.

5. Conclusion

This lab was definitely one of the more challenging ones, but starting early and working on it a little at a time made a huge difference. Once I got into the groove of testing pieces step by step, I was able to finish the design over a few days without feeling rushed or overly stressed. Along the way, I learned a lot about how pixels are generated and how a monitor actually gets its image from the timing signals and counters, instead of just treating VGA as a black box. It was especially cool to realize that this is very similar to how old-school games were implemented, where everything on screen was just rectangles and sprites mapped directly to pixel coordinates and updated frame by frame. Seeing the trains move, the player slide between lanes, the energy bar react, and the score update all from logic that I wrote myself made the project feel more like a real game than just a lab. Overall, it was rewarding to see that I could build something like this from scratch, all the way from basic signals to a complete interactive display.

6. Appendix

6.1 Code and Schematics

- ➔ **PixelAddress**
- ➔ **Player State Machine**
- ➔ **Train State Machine**
- ➔ **Display**
- ➔ **Top**