Lab 6 Report

Nate Jarrett
Lab Section 3

Description

The goal of this lab was to make a single-player game called Subway Slugging. This is essentially a bare bones clone of the game Subway Surfers. The basic game logic is as follows. Once the game start button has been pressed, trains start descending, starting with the left lane, then the right, and lastly the middle lane. On each track, there are at most 2 trains, with a new train being spawned after the previous train hits a trigger point on the screen. The trains have random lengths and wait a random delay after the trigger has been reached. The player can move left or right between the lanes to avoid the trains. If the player hits a train, then the game ends, and the reset button will have to be pressed. If the player is in the middle state and they press the jump button, they can activate a jump to go over the trains. This will drain their energy. If the energy runs out, then the character will become vulnerable again. The energy will begin recharging again once the jump button stops being pressed. After a train has passed the top of the player, the player will gain 1 point to their score. This continues until the player collides with a train.

Design

Pixel Address

The pixel address module is a pair of counters that increment across every row and column of the display and return the current coordinates. These counters are reset whenever they reach the end of the display boundaries. The value of these boundaries is 799 for the column and 529 for the rows. The active region is high so long as the column is less than 639 and the row is less than 479. Since the pixel values can never be negative, it is also constrained by 0 on its lower bound.

```
module pixel_address(
    input clk_i,
    output [15:0] row_o,
    output [15:0] col_o,
    output Hsync_o,
    output Vsync_o,
    output active_region_o
    );

    wire ovfl_col, ovfl_row;

    assign ovfl_col = col_o > 16'd799;
    assign ovfl_row = row_o > 16'd529;

    countUD16L columns (.clk_i(clk_i), .up_i(1'b1), .dw_i(1'b0), .ld_i(ovfl_col), .Din_i(16'b0), .Q_o(col_o));
    countUD16L rows (.clk_i(clk_i), .up_i(ovfl_col), .dw_i(1'b0), .ld_i(ovfl_row), .Din_i(16'b0), .Q_o(row_o));

    assign active_region_o = ((row_o >= 16'd0) & (row_o <= 16'd479)) & ((col_o >= 16'd0) & (col_o <= 16'd639));
endmodule
```

Figure 1: Pixel Address Module

Syncs

The sync module checks if the current pixel address is within the specified Hsync or Vsync range of the VGA display. These are predefined in the lab specification as 489-490 for Vsync and 655-750 for Hsync. The syncs for the VGA display were originally handled in the pixel address module; however, after further thought, I decided to make it into its own module. This is because the syncs may not always be needed when the pixel address counter is so separating the two is more practical.

```
module sync(
    input [15:0] row_i,
    input [15:0] col_i,
    input clk_i,
    output Hsync_o,
    output Vsync_o
    );

    FDRE #(.INIT(1'b0)) h_sync (.C(clk_i), .R(1'b0), .CE(1'b1), .D(h_sync_n), .Q(Hsync_o));
    FDRE #(.INIT(1'b0)) v_sync (.C(clk_i), .R(1'b0), .CE(1'b1), .D(v_sync_n), .Q(Vsync_o));

    assign h_sync_n = (col_i < 16'd655) | (col_i > 16'd750);
    assign v_sync_n = (row_i < 16'd489) | (row_i > 16'd490);
endmodule
```

Figure 2: Display Sync Module

Player

The player module combines the player state machine with additional logic to simplify the output of the player's position and jumping status. To synchronise the button inputs, they are passed through edge detectors so there is no glitching when being passed to the state machine.

The player itself is supposed to move at 2 pixels per frame, so a new signal called double frame is created. In addition to the frame input, it checks if the current row is halfway between the start and the Vsync range. To handle the moving left and right, it checks if the current character position is less than the target position, and if so, it moves right. The opposite is true for moving left. The arrived signal is if the current position is equal to the target position. The character will only move if the player has not crashed. These signals are passed into the counter that increments/decrements the current position.

The last signal of note is the output signal for the character, which tells the RGB selector that the player's color should be displayed. This just checks if the current pixel is within the character's bounds. Char_left is the only value that is being moved, and since the character's width is a set value, we can define char_right as char_left + 16.

```
// Character Logic ----------------------------------------------
assign double_frame = frame_i | ((row_i == 16'd245) & (col_i == 16'd320)); // High twice per frame
assign move_left = (char_left > target_player_left) & ~crashed_i & double_frame;
assign move_right = (char_left < target_player_left) & ~crashed_i & double_frame;
assign arrived = char_left == target_player_left;

wire L_i, R_i;
edge_detector left (.clk_i(clk_i), .button_i(btnL), .edge_o(L_i));
edge_detector right (.clk_i(clk_i), .button_i(btnR), .edge_o(R_i));

// Moving character left and right ------------------------------
countUD16L character_left (.clk_i(clk_i), .up_i(move_right), .dw_i(move_left), .ld_i(INITIALIZER), .Din_i(16'd515), .Q_o(char_left));

// Player State Machine -----------------------------------------
player_fsm player (
    .clk_i(clk_i),
    .left_i(L_i),
    .right_i(R_i),
    .up_i(btnU),
    .arrived_i(arrived),
    .has_energy_i(has_energy_i),
    .collision_i(crashed_i),
    .player_left_o(target_player_left),
    .hover_o(hovering_o)
);

// Player Display -----------------------------------------------
wire [15:0] char_top;
assign char_top = 16'd360;
assign character_o = ((col_i >= char_left) & (col_i <= char_left + 16'd16)) & // x-axis character bounds
                     ((row_i >= char_top) & (row_i <= char_top + 16'd16)); // y-axis character bounds
```

Figure 3: Player Module

Player State Machine

State Diagram

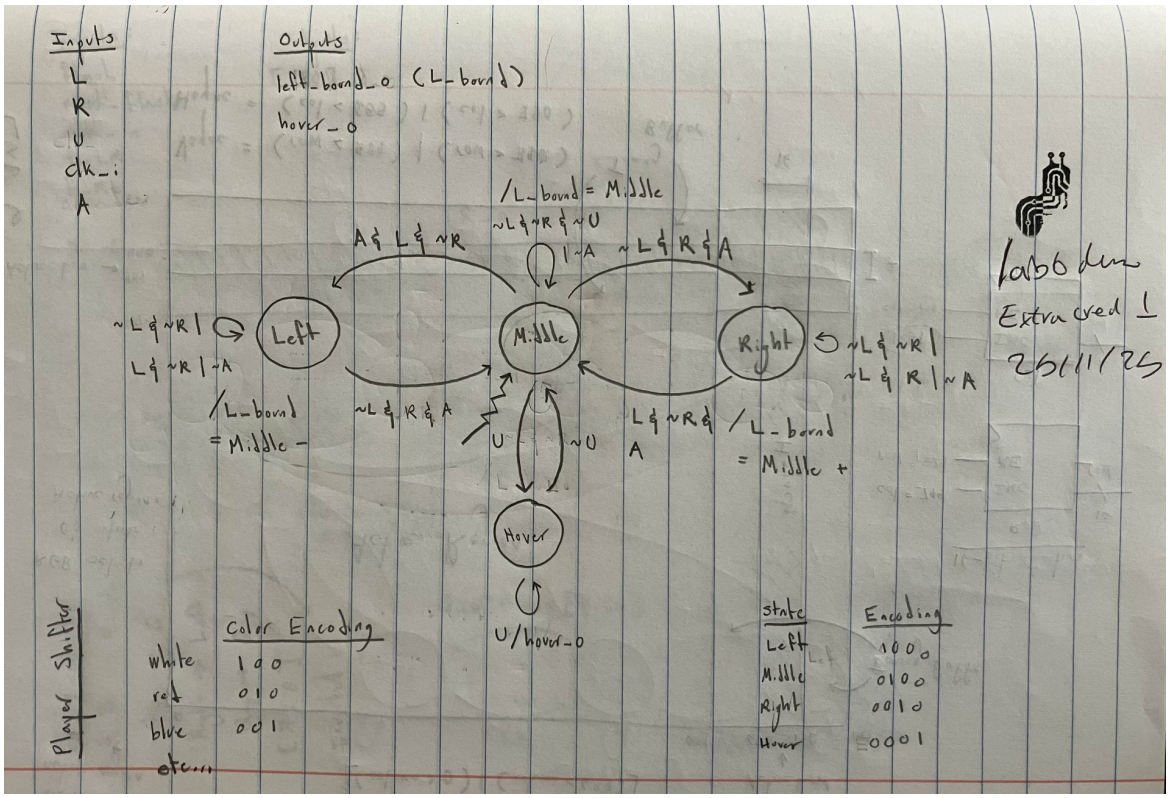

Figure 4: Player FSM Diagram

Logic Equations

State	Encoding	Logic Equation
Middle	1000	(M_s & ~left_i & ~right_i & ~up_i) | (M_s & ~arrived_i) | (H_s & ~up_i) | (L_s & ~left_i & right_i & arrived_i) | (R_s & left_i & ~right_i & arrived_i)
Left	0100	(M_s & left_i & ~right_i & arrived_i) | (L_s & ~left_i & ~right_i) | (L_s & left_i & ~right_i) | (L_s & ~arrived_i)
Right	0010	(M_s & ~left_i & right_i & arrived_i) | (R_s & ~left_i & ~right_i) | (R_s & ~left_i & right_i) | (R_s & ~arrived_i)
Hover	0001	(M_s & up_i & arrived_i & ~collision_i) | (H_s & up_i)

Table 1: Player FSM Logic Equations

Output Equations

Output	Logic Equation
Player_Left_o	(MIDDLE & {16{M_s | H_s}}) | (LEFT & {16{L_s}}) | (RIGHT & {16{R_s}})
Hover_o	H_s

Table 2: Player FSM Output Equations

```
assign MIDDLE = 16'd515;
assign LEFT = MIDDLE - 16'd70;
assign RIGHT = MIDDLE + 16'd70;
```

The MIDDLE value is a hardcoded value for the left bound of the character while in the middle lane. Since the tracks are a fixed distance apart, we can define the LEFT and RIGHT values based on the MIDDLE value. This allows for the ease of moving the character.

Description

Middle is the initial state for the finite state machine (FSM). While in this state, the FSM waits for a directional input from either button L, button R, or button U. If button L is pressed, then the FSM will move to the Left state. If button R is pressed, it will move to the Right state, and if button U is pressed, then it will move to the Hover state.

In the Left state, the FSM will first wait until the player has reached the left lane before being able to move again. This condition is defined by the arrived_i input, which checks the current player position against the target position. Once the player has arrived at the left lane, then it is free to move back to the middle lane, but until then, no inputs will have any effect except for the global reset. If the left button is pressed again or if no button is pressed, then the FSM will remain in the Left state.

In the Right state, the FSM follows the same arrival logic as the left state. The only difference is that it waits for the plater to arrive in the right lane. Once the player has arrived at the right lane, they are free to move back into the middle lane. If the right button is pressed again or if no button is pressed, then the FSM will remain in the Right state.

The Hover state is maintained so long as button U is being pressed. The player can only enter the hover state from the middle state. Originally, the FSM would exit the Hover state when the player ran out of energy, but in the final version, the logic for jumping is handled externally. While in the Hover state, it outputs the signal that the player is attempting to hover, regardless of the current energy level. This is combined with the energy bar module to determine if the player is invulnerable.

Game State Machine

State Diagram

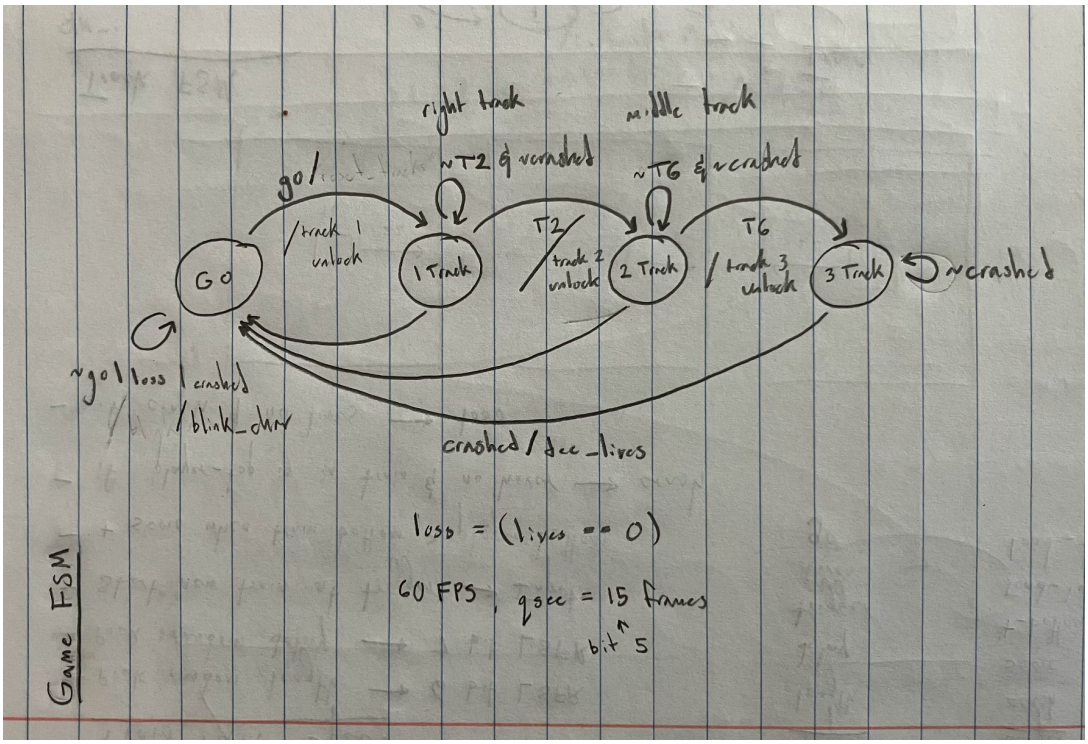

Figure 5: Game FSM Diagram

Logic Equations

State	Encoding	Logic Equation
Go	1000	(GO_s & ~go_i) | (GO_s & loss_i) | (ONE_s & crashed_i) | (TWO_s & crashed_i) | (THREE_s & crashed_i)

1 Track	0100	(GO_s & go_i & ~loss_i) | (ONE_s & ~T2_i & ~crashed_i)
2 Track	0010	(ONE_s & T2_i & ~crashed_i) | (TWO_s & ~T6_i & ~crashed_i)
3 Track	0001	(TWO_s & T6_i & ~crashed_i) | (THREE_s & ~crashed_i)

Table 3: Game FSM Logic Equations

Output Equations

Output	Logic Equation
Track_1_Unlock	GO_s & go_i & ~loss_i
Track_2_Unlock	TWO_s & T6_i
Track_3_Unlock	ONE_s & T2_i

Table 4: Game FSM Output Equations

Description

Go is the initial state for the FSM. It will stay in the Go state until the go signal (button C) is given. It will also never leave the Go state once the player has crashed. Once the go signal is given, it will move to the 1 Track state and unlock track 1.

While in the 1 Track state, it will remain there until 2 seconds have passed. This timer is handled by an instance of the timer module that resets when the go signal is high or on the transition between 1 Track and 2 Track. If the player crashes in this state, it will get sent back to the Go state.

After 2 seconds, the FSM moves to the 2 Track state, where it will remain until another 6 seconds have passed. On the transition, it will unlock track 2 and output that signal. This allows for the tracks to be initially staggered, which gives more randomness to the game. Once again, if the player crashes, they get sent back to the Go state, where they will stay.

After 6 more seconds, the FSM unlocks track 3 and moves to the 3 Track state, where it will remain until the player crashes or the global reset is given. Once the player crashes, it will get sent back to the Go state. I initially made an attempt to have multiple lives, but the logic was not complete in time.

Energy Bar

The energy bar module handles all the logic for calculating the player's energy and displaying it. The energy level will increment/decrement once per frame. The energy will only increase if the player is not in the hover state and the current energy is less than 192 (the maximum). The energy will only decrease if the player is in the hover state and the current energy is greater

than 0. If the energy is within this range, then the has_energy signal is high. The display output follows the same format as usual, with it being high between 0 and the current energy.

The initializer flip-flop was my method of initializing the energy bar to be at its maximum value at the start of the game. It is just a flip-flop with an init of 1, and after the first clock cycle, it is 0 for the rest of the time. This value is passed into the load data input for the counter, which will load it to its maximum. Without this, the energy would start at 0, then regenerate up to its max. This same method was used for the starting player character position.

```
// Initializer
wire INITIALIZER;
FDRE #(.INIT(1'b1)) initializer (.C(clk_i), .R(1'b0), .CE(1'b1), .D(1'b0), .Q(INITIALIZER)); // Initial reset state

// Energy Position ------------------------------------------------------------------------
wire [15:0] energy_bar_left, energy_bar_bottom;
assign energy_bar_left = 16'd10; // Test Location
assign energy_bar_bottom = 16'd202 & {16{has_energy_o}}; // Test location

// Energy Level ---------------------------------------------------------------------------
wire [15:0] energy_capacity, curr_energy;
wire energy_up, energy_dw;
countUD16L energy_bar_counter (
    .clk_i(clk_i),
    .up_i(energy_up & frame_i),
    .dw_i(energy_dw & frame_i),
    .ld_i(INITIALIZER),
    .Din_i(16'd192), // Energy Starts Full
    .Q_o(curr_energy)
);

assign energy_up = ~hover_i & (curr_energy < 16'd192);
assign energy_dw = hover_i & (curr_energy > 16'd0);
assign has_energy_o = (curr_energy > 16'd0) & (curr_energy <= 16'd192);
assign energy_capacity = curr_energy & {16{has_energy_o}};

// Energy Display -------------------------------------------------------------------------
assign energy_bar_o = ((col_i >= energy_bar_left) & (col_i <= energy_bar_left + 16'd20)) & // x-axis energy bounds
                      ((row_i <= energy_bar_bottom) & (row_i >= energy_bar_bottom - energy_capacity)); // Energy level
```

Figure 6: Energy Bar Module Snippet

Timer

There are 2 main components to the timer. Some signals allow for the flashing of the different components for either a quarter second or a half second. They alternate between high and low in their respective periods by using the frame counter signal. Since the program runs at 60 frames per second, a quarter second is 15 frames, and a half second is 30 frames. We can just take the 5th and 6th bits of the current frame to see when they alternate, and those are our flashing signals.

There are also the signals that output when 2 seconds (T2) or 6 seconds (T6) have passed. The reason they are separated is the fact that T2 will stay high after 2 seconds have passed, which does not work with flashing. The same goes for T6. The counter for those signals takes the 3rd bit of the current frame, which alternates every ⅛ of a second, and passes it through an edge detector. This makes it so it increments every quarter second, as the edge detector only goes high on the positive edge.

```
module timer(
    input clk_i,
    input frame_i,
    input reset_game_i,
    output qsec_o,
    output hsec_o,
    output T2_o,
    output T6_o
    );

    // Time Counter ------------------------------------------------------
    wire [15:0] frame_count, quarter_seconds;
    wire qsec_edge;

    countUD16L frame_counter (.clk_i(clk_i), .up_i(frame_i), .dw_i(1'b0), .ld_i(reset_game_i), .Din_i(16'd0), .Q_o(frame_count)
    );

    assign qsec_o = frame_count[4]; // High for quarter second, low for quarter second
    assign hsec_o = frame_count[5]; // High for half second, low for half second

    edge_detector quarter_sec_detector (.clk_i(clk_i), .button_i(frame_count[3]), .edge_o(qsec_edge));

    countUD16L quarter_sec_counter (.clk_i(clk_i), .up_i(qsec_edge), .dw_i(1'b0), .ld_i(reset_game_i), .Din_i(16'd0), .Q_o(quarter_seconds));
    assign T2_o = (quarter_seconds >= 16'd8); // 2 seconds
    assign T6_o = (quarter_seconds >= 16'd24); // 6 seconds
endmodule
```

Figure 7: Timer Module

Train

The train module is used to create a new train on any of the tracks. It takes in a few different values: go, started, advance, and T_i. Values such as the clock, frame, and current pixel are also included, but those are not unique. The go signal is used to load the initial train position at the top of the screen and to begin the delay countdown. The started signal is attached to a latch, so once the train has been started, it will remain high. This was initially used as an attempt to debug the lives implementation, but in the final version, it does not make much of an impact. The advance signal is provided by the RGB Selector module and is dependent on whether the player has crashed or not. This makes it so that all trains stop once there has been a collision.

The random delay is created by the track module; however, to count the delay, it is handled within each train. The random delay is passed in as an input and is latched into several flip-flops that store the target delay value. As the current delay counter increments, it checks itself against the target value. Once the target value has been reached, it latches a 1 into the flip-flop for the delayed go signal. This delayed go signal is used as an input to advance the movement counter for the bottom of the train.

The length of the train is randomly sampled through the use of an LSFR. The number sampled is a 6-bit value taken from the 6 least significant bits of the LSFR. This value is then added by 60 to get the actual length of the train. This ensures that the trains are never smaller than 60 pixels and never larger than 123 pixels.

There are a few additional signals that are outputted. If the top of the train reaches 360 (which is the top of the player character), it outputs a signal to increment the score. If the bottom of the train reaches the specified trigger value, it outputs the trigger value for the next train to begin is delay countdown. Finally, it outputs the trademark display signal to signify if the current pixel is within the train to be used for color selection.

```
// Started Latch ------------------------------------------------------------------------
wire started;
FDRE #(.INIT(1'b0)) begun (.C(clk_i), .R(reset_game_i), .CE(start_i), .D(1'b1), .Q(started));

// Train Length Calculation -------------------------------------------------------------
assign random_length[15:6] = 10'b0; // Upper bits zeroed
lsfr length_randomizer (.clkin(clk_i), .q_o(rnd));
FDRE #(.INIT(6'b0)) train_length [5:0] (.C(clk_i), .R(1'b0), .CE(go_i), .D(rnd[5:0]), .Q(random_length[5:0]));
assign train_length_o = 16'd60 + random_length; // Train length between 60 and 123 pixels

// Train Delay Calculation --------------------------------------------------------------
wire [15:0] curr_delay, delay_target;
countUD16L delay_counter (.clk_i(clk_i), .up_i(frame_i), .dw_i(1'b0), .ld_i(go_i & frame_i), .Din_i(16'd0), .Q_o(curr_delay));

wire delay_go;
assign delay_reached = (delay_target[6:0] <= curr_delay[6:0]);
FDRE #(.INIT(1'b0)) delay_latch (.C(clk_i), .R(go_i), .CE(delay_reached), .D(1'b1), .Q(delay_go));
FDRE #(.INIT(7'b0)) delay_target_latch [6:0] (.C(clk_i), .R(1'b0), .CE(go_i), .D(T_i[6:0]), .Q(delay_target[6:0]));

// Train Movement -----------------------------------------------------------------------
countUD16L train_bottom (.clk_i(clk_i), .up_i(advance_i & frame_i & started & delay_go), .dw_i(1'b0), .ld_i(go_i), .Din_i(16'd7), .Q_o(train_bottom_o));

// Train Display ------------------------------------------------------------------------
wire negative;
assign negative = (train_bottom_o < train_length_o);

assign track_right_i = track_left_i + 16'd60;
assign train_top_o = ((train_bottom_o - train_length_o) & {16{~negative}}) |
                     (16'd0 & {16{negative}}); // Prevent underflow
assign active_train_o = started & ((col_i >= track_left_i) & (col_i <= track_right_i)) &
                                  ((row_i <= train_bottom_o) & (row_i >= train_top_o));
assign trigger_o = (train_bottom_o == trigger_val_i) & frame_i;
assign score_inc_o = (train_top_o == 16'd360) & frame_i;
```

Figure 8: Train Module Snippet

Track

The track module handles the logic for creating the delays and creating instances of the trains. The delays are made by an LSFR, each storing a 7-bit number. The random delay is updated every frame, but the delays themselves are saved within the train modules. This allows for 2 different delays simultaneously for each of the trains.

Along with the random delay, there are two instances of the train module. The majority of the inputs and outputs are the same. The difference in the input signals is the go and start signals. In the first train, it takes the go signal from the FSM and the trigger from train 2 as its go signal. In the second train, it takes only the trigger from train 1 as its go signal. The start signals follow the same pattern. Each train outputs its own respective trigger, score increment, and display signals.

```
module track(
        input [15:0] trigger_i,
        input [15:0] track_left_i,
        input [15:0] col_i,
        input [15:0] row_i,
        input clk_i,
        input advance_i,
        input frame_i,
        input go_i,
        input reset_game_i,
        input crashed_i,
        output train_1_o,
        output train_2_o,
        output score_inc_o
    );

    wire [15:0] random_delay;
    wire [6:0] rnd;
    wire train_1_trigger, train_2_trigger;

    // Train Delay Calculation ----------------------------------------------------------
    assign random_delay[15:7] = 9'b0; // Upper bits zeroed
    lsfr delay_randomizer (.clkin(clk_i), .q_o(rnd));
    FDRE #(.INIT(7'b0)) train_delay [6:0] (
        .C(clk_i),
        .R(1'b0),
        .CE(frame_i),
        .D(rnd[6:0]),
        .Q(random_delay[6:0])
    );
```

Figure 9: Track Module Snippet

RGB Selector

Collisions

The collision logic has 2 main components.

1) It checks if both the display signal for the character and the display signal for any of the trains are high at the same time. If so, then there has been a collision.
2) It checks if the character is currently not hovering or is hovering but doesn't have any energy. Otherwise, the character cannot crash. Switch #3 also acts as a cheat code switch, so when it is on, the character is immortal.

```
// Collision Logic ------------------------------------------------------
assign collision = ((character & (t1_train_1 | t1_train_2 | t2_train_1 | t2_train_2 | t3_train_1 | t3_train_2))) &
                   (~hover_i | ~has_energy_o) & ~sw[3];
assign advance_val = reset_game ? 1'b1 : 1'b0;
FDRE #(.INIT(1'b1)) crash_detector (.C(clk_i), .R(1'b0), .CE(collision), .D(advance_val), .Q(advance));

assign crashed = ~advance;
```

Figure 10: RGB Collision Logic

Borders

The border logic simply creates 8 pixel-wide borders on all edges of the screen. It does this by outputting a high border signal if the current pixel is within 8 pixels of the edge, and then the color selector will handle displaying the corresponding color.

```
// Sectional Logic ------------------------------------------------------------
assign border = ((col_i >= 16'd0) & (col_i <= 16'd7)) | // Left Border
                ((col_i >= 16'd632) & (col_i <= 16'd639)) | // Right Border
                ((row_i >= 16'd0) & (row_i <= 16'd7)) | // Top Border
                ((row_i >= 16'd472) & (row_i <= 16'd479)); // Bottom Border
```

Figure 11: RGB Border Logic

Tracks / Score

There are 3 instances of the track module. Each instance takes in the predefined left bound of the track and the trigger for the track. Each instance then outputs each of its two train output signals along with the signal for incrementing the score.

```
assign track_1_trigger = 16'd400;
assign track_2_trigger = 16'd440;
assign track_3_trigger = 16'd400;
```

The score is handled with a counter that increments when any of the 3 tracks outputs an increment score signal.

Track Display

The track output signals take the hard-coded bounds of the tracks and add rails for the trains to visually move. These rails are 4 pixels wide and at both sides of the track.

```
// Track Display --------------------------------------------------------
wire track_1, track_2, track_3, train;
assign track_1 = ((col_i >= track_1_left) & (col_i <= track_1_left + 16'd4)) |
                 (col_i >= (track_1_left + 16'd56)) & (col_i <= (track_1_left + 16'd60));
assign track_2 = ((col_i >= track_2_left) & (col_i <= track_2_left + 16'd4)) |
                 (col_i >= (track_2_left + 16'd56)) & (col_i <= (track_2_left + 16'd60));
assign track_3 = ((col_i >= track_3_left) & (col_i <= track_3_left + 16'd4)) |
                 (col_i >= (track_3_left + 16'd56)) & (col_i <= (track_3_left + 16'd60));
assign train = t1_train_1 | t1_train_2 | t2_train_1 | t2_train_2 | t3_train_1 | t3_train_2;
```

Figure 12: RGB Track Rails Logic

```
assign track_1_left = track_2_left - 16'd70;
assign track_2_left = 16'd493;
assign track_3_left = track_2_left + 16'd70;
```

Colors

This logic handles selecting the color for each pixel depending on what is trying to be displayed. It first checks if the current pixel is in the active region; if not, then it sets all the values to 0's. For each item being displayed, it extends the signal, then "ands" it with its respective RGB value.

For the trains, it checks that none of the trains overlap with the character or the border. For the character to see if it's hovering or if it has crashed. If it's hovering, it will flash every half-second. If it has crashed, it will flash every quarter second. For the tracks, it checks that they don't overlap with the border, character, or trains.

```
// Color Logic ----------------------------------------------------------
assign red_o = {4{active_region_i}} & (({4{border}} & white[3:0]) | // White for border
               ({4{character}} & red[11:8] & {4{~hover_i | (hover_i & ~has_energy_o)}}) | // Red for character
               ({4{character}} & red[11:8] & {4{hover_i}} & {4{has_energy_o}} & {4{hsec}} & {4{~crashed}}) |
               ({4{track_1 | track_2 | track_3}} & white[11:8] & {4{~train}} & {4{~border}} & {4{~character}})) | // Yellow for hover
               ({4{~active_region_i}} & 4'b0); // All 0's
assign green_o = {4{active_region_i}} & (({4{border}} & white[7:4]) | // White for border
                 ({4{energy_bar}} & green[7:4]) | // Green for bar
                 ({4{character}} & green[7:4] & {4{~border}} & {4{hover_i}} & {4{has_energy_o}}) |
                 ({4{track_1 | track_2 | track_3}} & white[11:8] & {4{~train}} & {4{~border}} & {4{~character}})) | // Green for hover
                 ({4{~active_region_i}} & 4'b0); // All 0's
assign blue_o = {4{active_region_i}} & (({4{border}} & white[11:8]) | // White for border
                ({4{t1_train_1 | t1_train_2}} & blue[3:0] & {4{~border}} & {4{~character}}) | // Blue for Train
                ({4{t2_train_1 | t2_train_2}} & blue[3:0] & {4{~border}} & {4{~character}}) | // Blue for Train
                ({4{t3_train_1 | t3_train_2}} & blue[3:0] & {4{~border}} & {4{~character}}) | // Blue for Train
                ({4{character}} & blue[3:0] & {4{crashed}} & {4{qsec}}) |
                ({4{track_1 | track_2 | track_3}} & white[11:8] & {4{~train}} & {4{~border}} & {4{~character}})) | // Purple for crashed
                ({4{~active_region_i}} & 4'b0); // All 0's
```

Figure 13: RGB Color Selection Logic

These definitions for the colors aren't really the most important. They allow for changing of the exact color values, but since the 0's are omitted in the color logic, they are really just labels.

```
assign white = 12'b111111111111;
```

```
assign red = 12'b111100000000;
assign green = 12'b000011110000;
assign blue = 12'b000000001111;
```

Top

Frame Detector

The frame detector is an edge detector that takes the Vsync signal as its input. Since the Vsync only has one positive edge per frame, it acts as the perfect frame signal. The frame signal will stay high for one clock cycle per frame. This signal is then passed into a variety of modules that use it, so they only act once per frame.

Score Display

The score is displayed in the same way it has been displayed for the last few labs. It uses a ring counter that acts as the main controller. The selector then takes in the ring counter output to select the 4 bits that will be passed into the hex7seg module. This module converts the 4 bits into their hexadecimal representation on the 7-segment display. The 2 leftmost anodes are not used, so they are set to 1 since the display is active low. The remaining anodes will only go low when it is their turn to display based on the ring counter.

```
// Score Display Logic ----------------------------------------------------------------------
wire [15:0] display_o;
wire [3:0] ring, sel;

ring_counter round (.clk_i(clk), .advance_i(digsel), .ring_o(ring)); // Ring Counter, round and round we go
selector select (.Sel_i(ring), .N_i(display_o), .H_o(sel)); // Ring Counter -> Selector
hex7seg segment (.N_i(sel), .Seg_o(seg)); // Selector -> Hex 7 Display

assign an[3] = 1'b1;
assign an[2] = 1'b1;
assign an[1] = ~ring[1];
assign an[0] = ~ring[0];
```

Figure 14: Score Display Logic

Testing and Simulation

To ensure that my syncs were operating as intended, I used the provided test simulation. However, this testbench ended up being buggy, so it wasn't much help. My primary form of testing the rest of the modules was through generating the bit stream and checking the display itself. This is because a simulation would not be at a good time scale to do any meaningful analysis or debugging. A lot of incremental testing was done to check which component of a module might've been failing. Only after confirming a module is good to go would I move on to the next. This narrowed down any possible issues could be a great extent.

I did not end up doing any simulation testing for my finite state machines. I feel like I have gotten pretty good at designing complete FSMs that do not have are gaps in the logic. I had to do a few iterations for the player FSM to get the player not to move while in a transition, but this was simply adding another input that relates the current position to the target position. Other than that, there were no instances where the FSM was in no state or outputted the incorrect output signals. The same thing goes for the game FSM.

Results

This lab was the most time-consuming but not necessarily the most difficult. Each of the individual components of the lab is not that complicated; it just requires thinking about it the right way. Since this lab comes at the end of the course, I have the most experience and knowledge surrounding structural Verilog possible. Compared to an earlier lab, like lab 2 or 3, it was much easier to see where a fault in my logic might be. This kept me from getting stuck on one issue for multiple hours without any progress.

Conclusion

Overall, I feel like this lab went pretty smoothly. It took a very long time to implement all the different components of the lab, but each part didn't have many problems. The incremental design approach that was recommended by the lab description was very helpful. It allows me to continuously build in steps without trying to string everything together all at once. If I had to simultaneously debug the issue with my syncs, along with something like my player FSM, it would be 10 times harder. I ended up using the full 2 weeks to get this lab done, but I managed to get the lab done on time with extra credit. I'm not sure how I would speed up the process, but I know that I would like to when doing future labs/projects that are as big as this one. I would say that the hardest part of this lab was the train and track modules. This is because there were a lot of moving parts to take into account. Once again, the incremental approach was invaluable, so I could tackle first the moving train, then creating 2 instances, and finally adding the randomization. Trying to do these all at once would be a giant headache.

Appendix

Timing Report

The maximum clock frequency my design will run at is 25 MHz. This is because the output of the provided slow clock is 25 MHz. This clock on the board itself runs at 100 MHz, but this is only passed into the clock port of the slowing module.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	30.918 ns	Worst Hold Slack (WHS):	0.118 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	788	Total Number of Endpoints:	788	Total Number of Endpoints:	456

All user specified timing constraints are met.

Figure 15: Design Timing Summary

Name	Waveform	Period (ns)	Frequency (MHz)
sys_clk_pin	{0.000 5.000}	10.000	100.000
clk_out1_clk_wiz_0	{0.000 20.000}	40.000	25.000
clkfbout_clk_wiz_0	{0.000 5.000}	10.000	100.000

Figure 16: Clock Timing Report

Simulation Diagram

This is the simulation diagram for the VGA syncs. There was an issue with the simulation that made the good Hsync value slowly go out of sync with the actual Hsync value. This caused progressively more Oops as they get further apart. Since there was no problem with the actual implementation/display, this is a result of the simulation. If this were not the case, then the VGA cable would not display as intended.

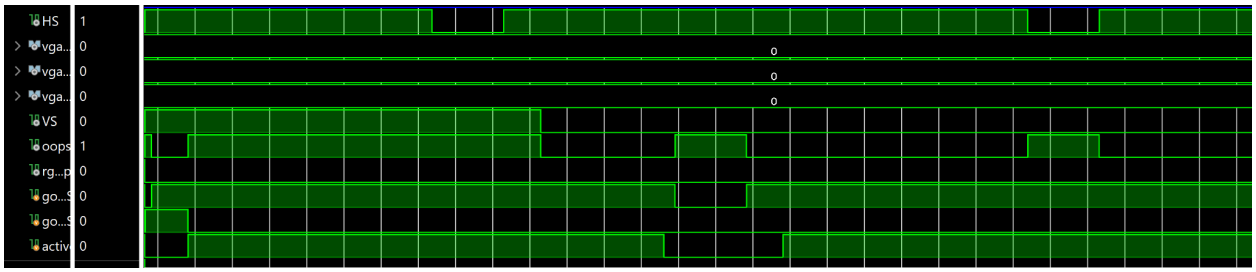

Figure 17: Vsync Edge Simulation