DLD PROJECT - MILESTONE 3 REPORT
Fall 2024

SPACE FORCE X

BY DISASTER STRIKES (Group 2)

Syed Amn Abbas Naqvi, Mahnoor Nauman Shaikh, Aneeza Khan & Syeda Seerat Zahra

Section: T1 [Farhan Khan / Khuzaima Ali Khan]

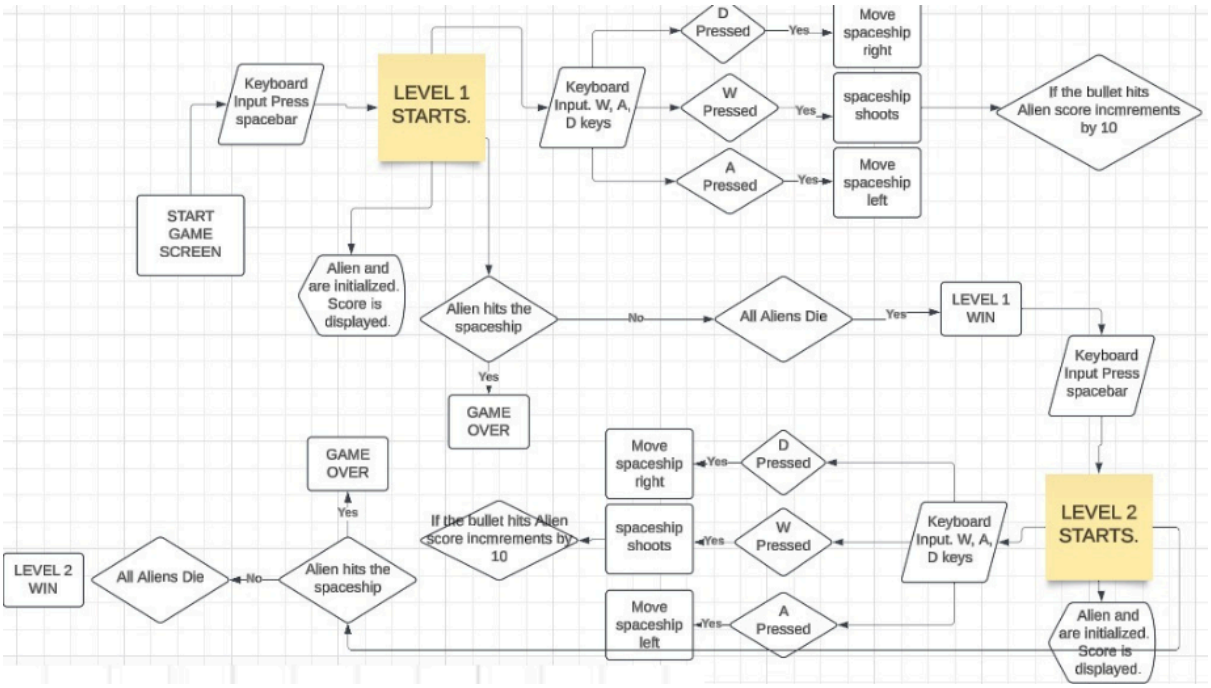
Abstract

Space Shooter is an engaging single-player game featuring a tank controlled by a keyboard. The player can move the tank left and right using the keyboard's A and D keys to navigate the battlefield and shoot at aliens descending downwards in columns by pressing the W key. When the W key is pressed, the tank fires bullets that can hit the aliens, increasing the player's score with each successful strike.

Using the P key the player can pause the game screen. The game can be reset using the U18 push button on the FPGA. However, the game ends if any alien collides with the tank, displaying a Game Over screen. The player wins Level 1 when all the aliens on the screen are killed and progress to Level 2 when the player presses the spacebar on the Level 1 Win screen. Level 2 has aliens appear in a different pattern and faster to increase the game's difficulty. The Game Over screen is displayed again if any alien collides with the tank. When all the aliens are killed the Level 2 Game Win screen is displayed.

Contents

1. Block Diagram



The block diagram above provides an overview of our game logic framework. Once the player presses the space bar on the Start Game Screen, Level 1 starts where the player will see the tank sprite and the pink aliens in their initial positions after which aliens will descend downwards, and through the keyboard inputs the player sprite will move left and right and fire bullets towards the aliens.

Collision detection with the bullet will eliminate the aliens and result in the score incrementing by 10. For Level 1 when the bullet collides with any alien in a column, the entire column of aliens is killed. Game over condition will occur if the alien collides with the tank after which the Game Over Screen will display. Once the player wins Level 1, they progress to the more challenging Level 2. The aliens now appear in tilted descending columns where the collision of one bullet only kills one alien. If any alien collides with the tank, the game ends. Once the player kills all the aliens on the screen, they win that level.

2. Input Block

2.1 Keyboard

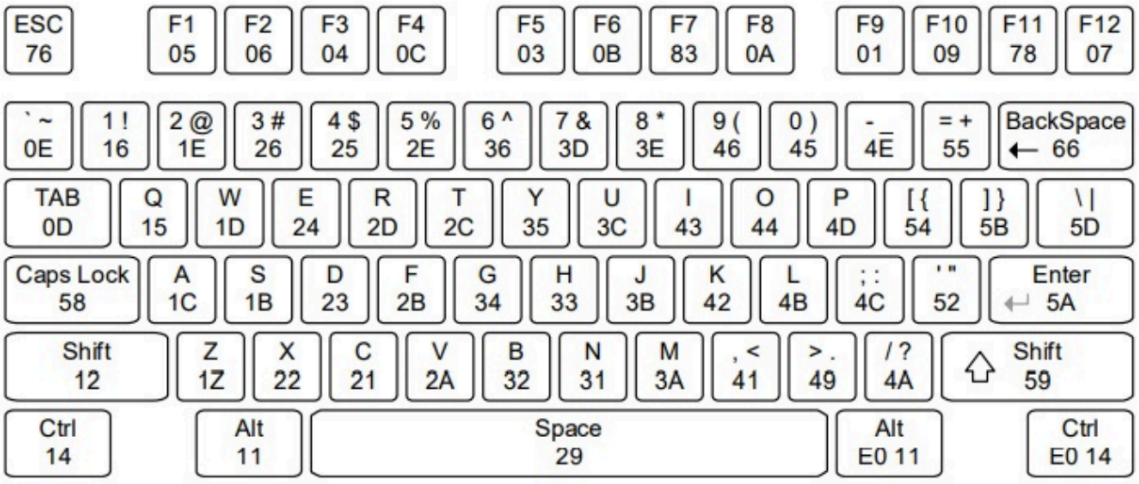
A keyboard is being used as our input peripheral for the game. The keys are used to move the player left/right and to shoot bullets towards the aliens as well as to pause the game. The game can be reset using the U18 push button of the FPGA. The PS/2 keyboard is interfaced with the Basys3 FPGA board. It utilizes the PS/2 communication protocol for sending data via the PS2D pin and receiving clock signals via the PS2C pin.

The keyboard_1 module processes PS/2 signals (ps2d and ps2c) and detects key presses. It includes outputs for various keys such as Space, arrow keys, and others, as well as a state signal and a key release flag. It instantiates a second module, ps2_rx, to handle the PS/2 communication protocol, which receives serial data from the keyboard and reconstructs it into 8-bit parallel data.

Within keyboard_1, the received data is stored in a register (key), and additional logic determines if a key release has occurred by checking for a specific release code (0xF0). Key-specific outputs are assigned based on the detected key codes, ensuring they are active only when the corresponding key is pressed and not released.

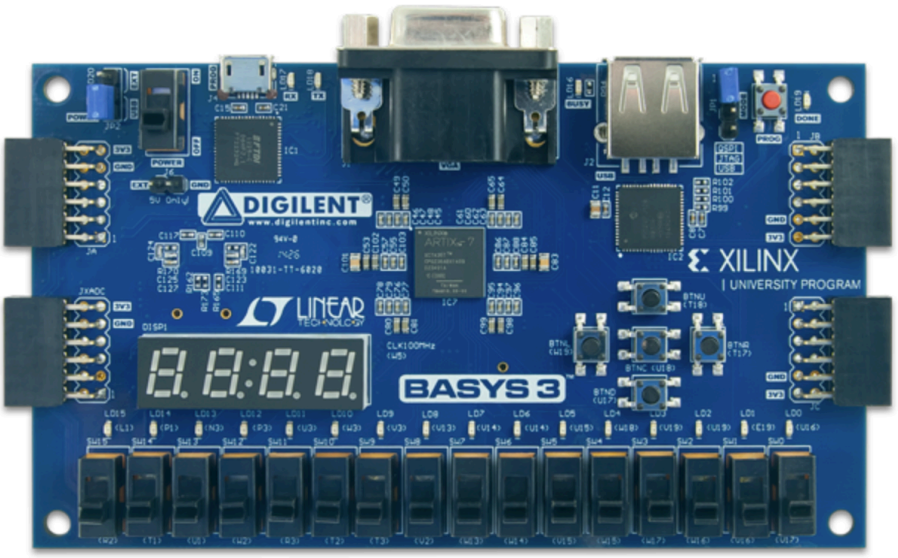
The following keys are used:

- W - Shoot
- A - Left/Move Left
- D - Right/Move Right
- Spacebar - Game Start + Level Transition
- P - Pause



2.2 Basys 3 Artix 7 FPGA

We are using the U18 push button on the FPGA for the reset game condition.



3. Control Block

The main modules implemented for the gameplay are as follows:

- Aliens Movement
- Bullets Movement
- Tank Control
- Text Generation
- Ascii_rom
- Game State FSM

3.1 Aliens Movement Modules

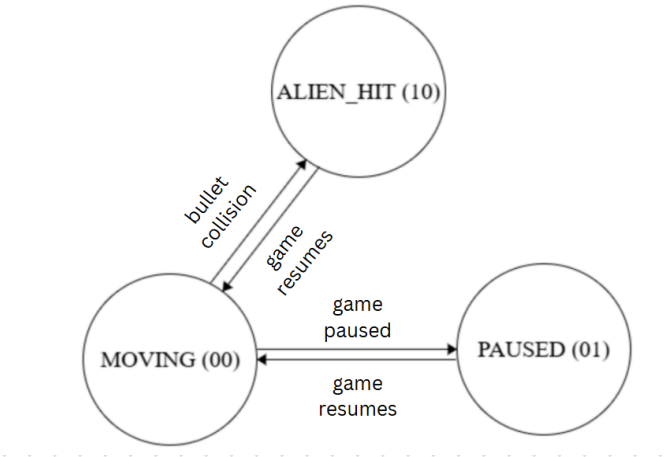
- These modules make use of a Moore FSM (the outputs are dependent on the current state) to control the movement of the aliens and handle the collision of aliens with bullets based on three states: Moving, Paused, and Alien_Hit.

- It renders the alien sprites, initializes aliens in different patterns according to the levels (explained difference between the two aliens_movement modules below), and handles alien movement if it is in the Moving state and alien collision with a bullet if in the Alien_Hit state. The speed is controlled using a move_counter, when the move_interval is reached then aliens move downwards based on the move_speed.

- Movement is paused when the P key on the keyboard is pressed (Paused state) and when it is released the state transitions from Paused to Moving state again.

- When the bullet collides with the aliens, the aliens transition from a moving state to an Alien_Hit state. It allows the alien to be marked as destroyed, and the game returns to the Moving state after the hit is processed.

- State Transition Diagram:



3.1.1 Aliens_Movement_Level1

5 columns of 4 aliens each, use column offsets to initialize alien columns in different positions. When an alien reaches the bottom of the screen it resets to the top of the screen. Since aliens are in vertical columns the collision logic is such that the whole column gets destroyed at once, making it an easier level.

3.1.2 Aliens_Movement_Level2

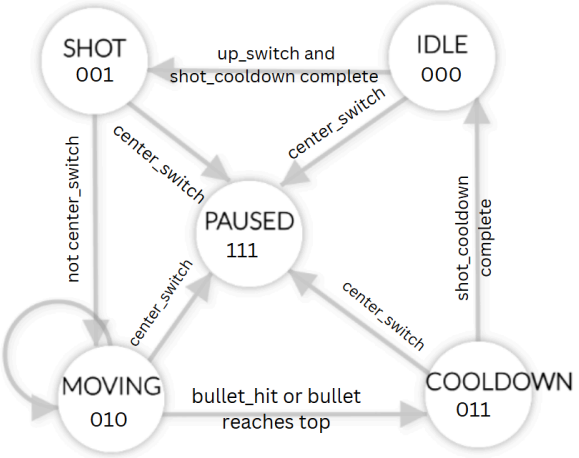
5 columns of 3 aliens each have aliens initialized as tilted columns, when an alien reaches the bottom of the screen it resets to the top. The positioning of the aliens makes it such that one alien at a time is destroyed by a bullet, adding difficulty to the gameplay.

3.2 Bullets Movement

- This module uses a Moore FSM to control the movement of the bullets using five states: Idle, Shot (creating new bullet), Moving (updating bullet position), Cooldown (delay after shot), and Paused. It also handles pixel rendering for the bullet.

- If the P key is pressed, any state can transition into the Paused state. The Idle state transitions to Shot if the shot cooldown period has passed and the W key is pressed. Shot to Moving state transition happens if not paused and Moving to Cooldown if no active bullets, else remain in the Moving state. In the Cooldown state after the counter for the cooldown delay has reached the shot_cooldown interval time it transitions to the Idle state.

- State Transition Diagram:



3.3 Tank Control

- This module handles movement and pixel rendering for the tank.

- Y position of the Tank is fixed at screen_height-80 (near the bottom of the screen). A movement_counter increments until it reaches the move_interval after which if the A key is pressed then for the left movement 4 pixels are subtracted from the x position of the tank and if the D key is pressed then for the right movement 4 pixels are added to the x position.

To display the score the Text Generation and ascii_rom modules are used:

3.4 Text Generation

- This module generates a single character on a VGA display. It determines when the current pixel of a specific character is within the bounds of its fixed position and outputs a signal, enabling the rendering of that character.

- The Text Generation Module is instantiated for each character to be displayed on the screen.
- For each instance, the displayContents signal determines if the corresponding character should be rendered at the current pixel location.
- The asciiData output retrieves the actual pixel data for the character from the ascii_rom module.

3.5 ascii_rom

- This module stores and retrieves bitmap representations of ASCII characters.
- The rom_addr input specifies the character and its row.
- The data output sends the pixel row to a display system for rendering.

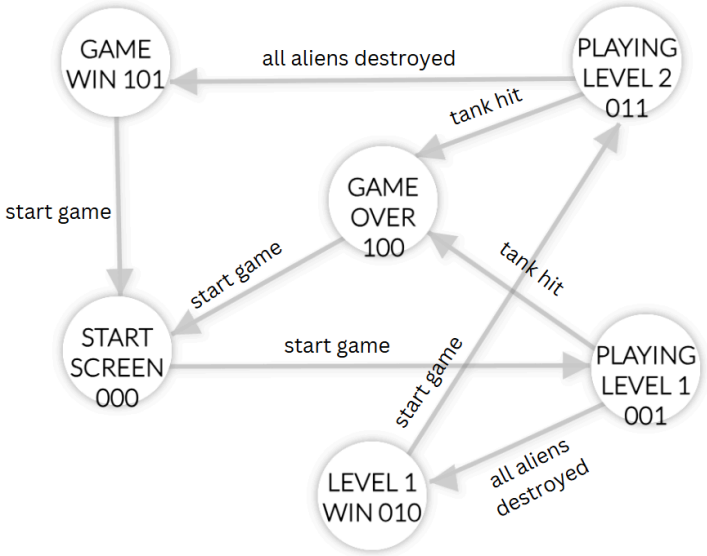
3.6 Game State FSM

- This is the main game state management FSM which has 6 states: Start_Screen, Playing_Level1, Level1_Win, Playing_Level2, Game_Over, and Game_Win. This is also a Moore machine.

- The signals tank_hit indicates the collision between alien and tank, all_aliens_destroyed signals that all aliens in the current level have been destroyed, start_game is triggered by the space bar to start the game and transition between levels, level_reset is an explicit signal for the aliens_movement modules to reset their state.

- When the game starts or the reset button is pressed the FSM goes to the Start Screen state, the current_level is set to 0 and the level_reset is set to 1 which resets the game components. When the space bar is pressed (start_game is true) then it transitions to Playing_Level1 which sets current_level to 1 and level_reset to 1. If tank_hit then transitions to Game_Over, if all_aliens_destroyed then it transitions to Level1_Win which also sets current_level to 2. If the spacebar is pressed (start game is true) then transition to Playing_Level2 which has the same game over (tank_hit) and game win (all_aliens_destroyed) state transition logic as Playing_Level2. Game_Over and Game_Win state reset current_level to 0.

- State Transition Diagram:

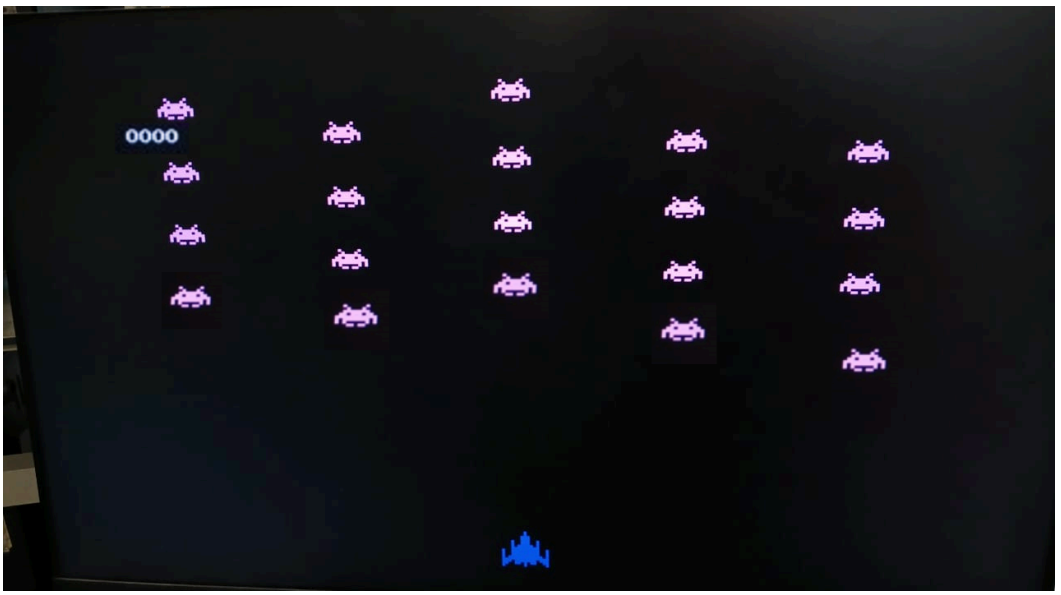


4. Output Block

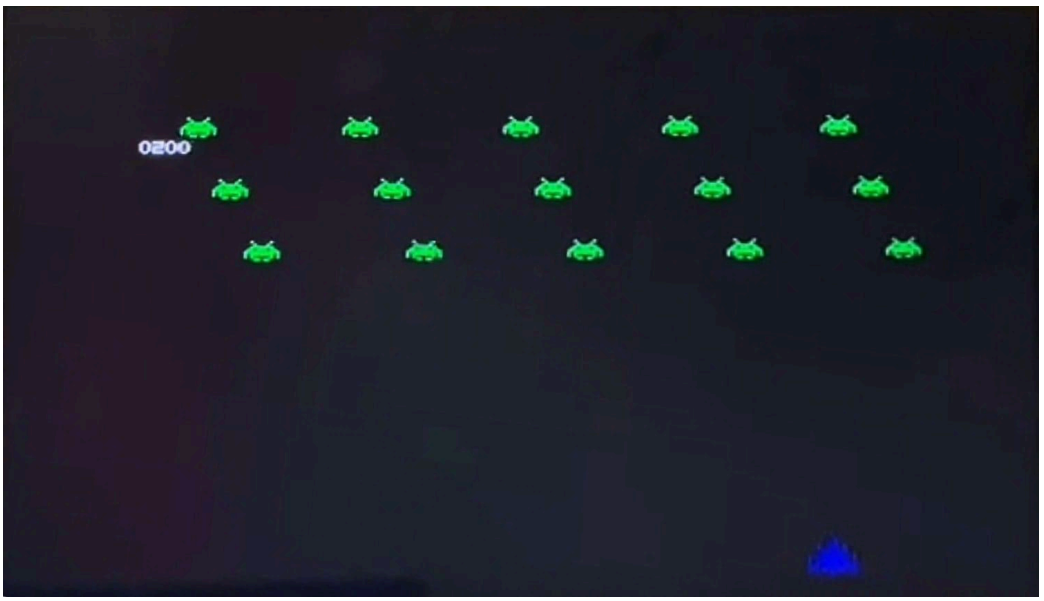
The display screen is generated using the VGA sync module and Top Level Module which connects and synchronizes the Alien Movement, Bullet Control, Tank Control, Text Generation and ascii_rom and game screen modules and determines pixel colors based on the game objects: aliens, tank, bullet, and background, and assigns them to the RGB outputs.

The screen used is the standard 640 x 480 pixels, video_on is turned on from 0 to 639 for the horizontal axis and 0 to 479 pixels for the vertical axis to create the main display screen.
The Start Screen is rendered using the start_bitmap_display (contains the bitmap for the Start Screen and handles pixel rendering) and start_screen (calls the bitmap display module and connects to vga sync module for output on the vga display) modules. Similarly for the other screens such as Level 1 Win, Level 2 Win, Game Over and Game Win screens there is a bitmap display module and then another module which calls it and connects to vga sync. All of these modules are then called in the TopLevelModule where according to the game mechanics the screen modules are called for pixel rendering.

Start Screen



Level 1 Screen



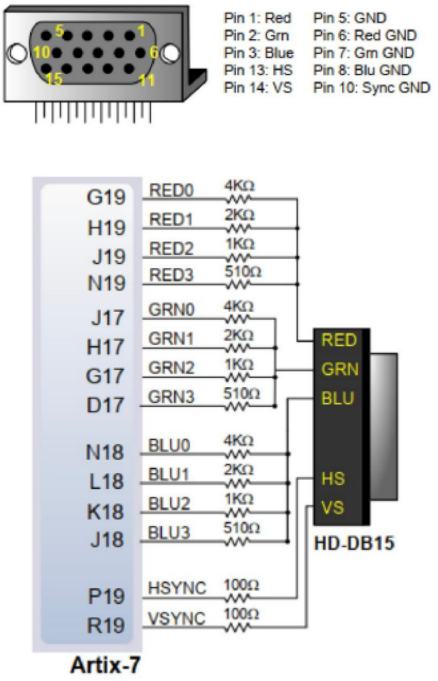
Level 2 Screen

Level 1 Win Screen

Level 2 Win Screen

Game Over Screen

The following pin configuration is used to connect the Top Level Module outputs to the FPGA:



5. Main Modules

Here are some of the main modules and functions we used for our project:

1. The Top Level module instantiates all the objects, screens, and states:

```
`timescale 1ns / 1ps
module TopLevelModule(
    input clk,                  // System clock
    input reset,                // Reset input
    input wire ps2d, ps2c,     // PS/2 keyboard data and clock
    output h_sync,              // Horizontal sync for VGA
    output v_sync,              // Vertical sync for VGA
    output [3:0] red,           // Red color output
    output [3:0] green,         // Green color output
    output [3:0] blue           // Blue color output
);

    // Internal wires
    wire clk_d;                 // Divided clock for
    wire [9:0] h_count;         // Horizontal pixel counter
```

```
wire [9:0] v_count;          // Vertical pixel counter
wire [9:0] pixel_x;          // Current pixel x-coordinate
wire [9:0] pixel_y;          // Current pixel y-coordinate
wire video_on;               // VGA video enable signal
wire trig_v;                // Vertical trigger signal for VGA sync

// Keyboard signals
wire SPACE, UP, DOWN, LEFT, RIGHT, PAUSE, RESET, SPECIAL;
wire key_state, key_release;   // State and release signals from the keyboard module

// Game object signals
wire alien_pixel_on;         // Alien pixel rendering signal
wire tank_pixel_on;          // Tank pixel rendering signal
wire bullet_pixel_on;        // Bullet pixel rendering signal
wire [9:0] tank_x;           // Tank position
wire bullet_active;          // Bullet active state
wire [9:0] bullet_x;         // Bullet x position
wire [9:0] bullet_y;         // Bullet y position
wire bullet_collision_confirmed; // Collision confirmation signal
wire bullet_hit;             // Wire to track bullet hitting an alien

reg [3:0] red_reg, green_reg, blue_reg; // Registers for RGB values

// Game state and level signals
wire [2:0] game_state;
wire [1:0] current_level;
wire level_reset;  // handle level resets
wire tank_hit;     // Signal to indicate tank is hit by alien
wire all_aliens_destroyed;  // Signal to indicate all aliens are destroyed

// Alien and level-specific signals
wire alien_pixel_on_level1, alien_pixel_on_level2;
wire bullet_collision_confirmed_level1, bullet_collision_confirmed_level2;
wire tank_hit_level1, tank_hit_level2;

// Instantiate the keyboard module
keyboard_1 k1 (
    .clk(clk),
    .reset(reset),
    .ps2d(ps2d),
    .ps2c(ps2c),
    .SPACE(SPACE),
    .UP(UP),
```

```
        .DOWN(DOWN),
        .LEFT(LEFT),
        .RIGHT(RIGHT),
        .PAUSE(PAUSE),
        .RESET(RESET),
        .SPECIAL(SPECIAL),
        .state(key_state),
        .key_release(key_release)
    );

    // Clock divider
    clk_div clkDivider (
        .clk(clk),
        .clk_d(clk_d)
    );

    // Horizontal counter
    h_counter horizontalCounter (
        .clk(clk_d),
        .h_count(h_count),
        .trig_v(trig_v)
    );

    // Vertical counter
    v_counter verticalCounter (
        .clk(clk_d),
        .enable_v(trig_v),
        .v_count(v_count)
    );

//    wire h_sync_wire;
//    wire v_sync_wire;
    // VGA sync module
    vga_sync vgaSync (
        .h_count(h_count),
        .v_count(v_count),
        .h_sync(h_sync),
        .v_sync(v_sync),
        .video_on(video_on),
        .x_loc(pixel_x),
        .y_loc(pixel_y)
    );
```

```
// Score register (initialize to 0)
reg [8:0] game_score = 9'd0;

wire [3:0] thousands, hundreds, tens, ones;
wire [6:0] a[3:0]; // Holds ASCII data for each digit
wire d[3:0];       // Display signals for each digit
wire score_display_on;

// Calculate digit values from game_score
assign thousands = (game_score / 1000) % 10;
assign hundreds  = (game_score / 100) % 10;
assign tens      = (game_score / 10) % 10;
assign ones      = game_score % 10;

// Text Generation Modules for Score Digits
textGeneration score_thousands (
    .clk(clk),
    .reset(reset),
    .asciiData(a[0]),
    .ascii_In(thousands + 7'h30),
    .x(pixel_x),
    .y(pixel_y),
    .displayContents(d[0]),
    .x_desired(10'd80),
    .y_desired(10'd80)
);

textGeneration score_hundreds (
    .clk(clk),
    .reset(reset),
    .asciiData(a[1]),
    .ascii_In(hundreds + 7'h30),
    .x(pixel_x),
    .y(pixel_y),
    .displayContents(d[1]),
    .x_desired(10'd88),
    .y_desired(10'd80)
);

textGeneration score_tens (
    .clk(clk),
    .reset(reset),
    .asciiData(a[2]),
```

```
    .ascii_In(tens + 7'h30),
    .x(pixel_x),
    .y(pixel_y),
    .displayContents(d[2]),
    .x_desired(10'd96),
    .y_desired(10'd80)
);

textGeneration score_ones (
    .clk(clk),
    .reset(reset),
    .asciiData(a[3]),
    .ascii_In(ones + 7'h30),
    .x(pixel_x),
    .y(pixel_y),
    .displayContents(d[3]),
    .x_desired(10'd104),
    .y_desired(10'd80)
);

// Score display logic
wire score_display_pixel;
wire [3:0] digit_pixels;
assign score_display_on = d[0] || d[1] || d[2] || d[3];
wire [6:0] score_ascii;
assign score_ascii = d[0] ? a[0] :
                     d[1] ? a[1] :
                     d[2] ? a[2] :
                     d[3] ? a[3] : 7'h30; // Default to '0'

// ASCII ROM for score display
wire [10:0] rom_addr;
wire [3:0] rom_row;
wire [2:0] rom_col;
wire [7:0] rom_data;
wire rom_bit;

ascii_rom score_rom (
    .clk(clk),
    .rom_addr(rom_addr),
    .data(rom_data)
);
```

```
    assign rom_row = pixel_y[3:0];
    assign rom_addr = {score_ascii, rom_row};
    assign rom_col = pixel_x[2:0];
    assign score_display_pixel = rom_data[~rom_col];

    // Score update and reset logic
    always @(posedge clk_d) begin
        if (reset) begin
            // Reset score to 0 when reset or level is reset
            game_score <= 9'd0;
        end else if (bullet_collision_confirmed) begin
            // Increment score when an alien is hit
            game_score <= game_score + 9'd10;
        end
    end

    // Level-specific tank hit detection
    assign tank_hit_level1 = (current_level == 2'b01) ? (alien_pixel_on_level1 &&
tank_pixel_on) : 1'b0;
    assign tank_hit_level2 = (current_level == 2'b10) ? (alien_pixel_on_level2 &&
tank_pixel_on) : 1'b0;

    // Consolidated tank hit signal
    assign tank_hit = tank_hit_level1 || tank_hit_level2;

    // All aliens destroyed logic
assign all_aliens_destroyed = (current_level == 2'b01 && game_score >= 9'd200) || //
Level 1 complete
                          (current_level == 2'b10 && game_score >= 9'd350); // Game complete

    // Alien movement modules for different levels
    aliens_movement_level1 alienCtrl_Level1 (
        .clk(clk_d),
        .reset(reset),
        .level_reset(level_reset),   // Map the level_reset signal from game_state_fsm
        .center_switch(PAUSE),
        .pixel_x(pixel_x),
        .pixel_y(pixel_y),
        .bullet_hit(bullet_active),
        .bullet_x(bullet_x),
        .bullet_y(bullet_y),
        .pixel_on(alien_pixel_on_level1),
        .bullet_collision_confirmed(bullet_collision_confirmed_level1)
```

```
    );

    aliens_movement_level2 alienCtrl_Level2 (
    .clk(clk_d),
    .reset(reset),
    .level_reset(level_reset),  // Map the level_reset signal from game_state_fsm
    .center_switch(PAUSE),
    .pixel_x(pixel_x),
    .pixel_y(pixel_y),
    .bullet_hit(bullet_active),
    .bullet_x(bullet_x),
    .bullet_y(bullet_y),
    .pixel_on(alien_pixel_on_level2),
    .bullet_collision_confirmed(bullet_collision_confirmed_level2)
);

    // Alien pixel and collision signals based on current level
    assign alien_pixel_on = (current_level == 2'b01) ? alien_pixel_on_level1 :
                            (current_level == 2'b10) ? alien_pixel_on_level2 :
                            1'b0;

    assign bullet_collision_confirmed = (current_level == 2'b01) ?
bullet_collision_confirmed_level1 :
                                        (current_level == 2'b10) ? bullet_collision_confirmed_level2 :
                                        1'b0;

    // Game State FSM
    game_state_fsm gameStateMachine (
        .clk(clk),
        .reset(reset),
        .tank_hit(tank_hit),
        .all_aliens_destroyed(all_aliens_destroyed),
        .start_game(SPACE),  // Start game with spacebar
        .game_state(game_state),
        .current_level(current_level),
        .level_reset(level_reset)
    );

    // Tank control module
    tank_control tankCtrl (
        .clk(clk_d),
        .reset(reset),
        .left_switch(LEFT),   // A to move LEFT
```

```
        .right_switch(RIGHT), // D to move RIGHT
        .pixel_x(pixel_x),
        .pixel_y(pixel_y),
        .pixel_on(tank_pixel_on),
        .tank_x_pos(tank_x)
    );

    // Bullet control module
    bullet_movement bulletCtrl (
        .clk(clk_d),
        .reset(reset),
        .pixel_x(pixel_x),
        .pixel_y(pixel_y),
        .up_switch(UP),          // W is used for single shot
        .center_switch(PAUSE),
        .tank_x(tank_x),
        .bullet_hit(bullet_collision_confirmed),  // Feedback from aliens module
        .pixel_on(bullet_pixel_on),
        .bullet_active(bullet_active),
        .bullet_x(bullet_x),
        .bullet_y(bullet_y)
    );

// Start Screen Module
    wire start_h_sync, start_v_sync;
    wire [3:0] start_red, start_green, start_blue;
    start_screen startScreen (
        .clk(clk),
        .reset(reset),
        .h_sync(start_h_sync),
        .v_sync(start_v_sync),
        .red(start_red),
        .green(start_green),
        .blue(start_blue)
    );

    // Level 1 Win Screen Module
    wire win1_h_sync, win1_v_sync;
    wire [3:0] win1_red, win1_green, win1_blue;
    level1_win_screen level1Win (
        .clk(clk),
        .reset(reset),
        .h_sync(win1_h_sync),
```

```
        .v_sync(win1_v_sync),
        .red(win1_red),
        .green(win1_green),
        .blue(win1_blue)
    );

    // Level 2 Win Screen Module
    wire win2_h_sync, win2_v_sync;
    wire [3:0] win2_red, win2_green, win2_blue;
    level2_win_screen level2Win (
        .clk(clk),
        .reset(reset),
        .h_sync(win2_h_sync),
        .v_sync(win2_v_sync),
        .red(win2_red),
        .green(win2_green),
        .blue(win2_blue)
    );

    // Lose Screen Module
    wire lose_h_sync, lose_v_sync;
    wire [3:0] lose_red, lose_green, lose_blue;
    lose_screen loseScreen (
        .clk(clk),
        .reset(reset),
        .h_sync(lose_h_sync),
        .v_sync(lose_v_sync),
        .red(lose_red),
        .green(lose_green),
        .blue(lose_blue)
    );

    reg [3:0] red_reg, green_reg, blue_reg;
    reg h_sync_reg, v_sync_reg; // sync registers

// Pixel Rendering
always @(posedge clk_d) begin
    if (~video_on) begin
            red_reg <= 4'h0;
            green_reg <= 4'h0;
            blue_reg <= 4'h0;
//            h_sync_reg <= h_sync_wire;
//            v_sync_reg <= v_sync_wire;
```

```
end
else begin
   case (game_state)
         3'b000: begin  // Start Screen
            h_sync_reg <= start_h_sync;
            v_sync_reg <= start_v_sync;
            red_reg <= start_red;
            green_reg <= start_green;
            blue_reg <= start_blue;
         end

      3'b001: begin  // PLAYING_LEVEL1 state
         if (alien_pixel_on) begin
            // Pink color for Level 1 aliens
            red_reg <= 4'hF;
            green_reg <= 4'h8;
            blue_reg <= 4'hF;
         end
         else if (tank_pixel_on) begin
            // Dark blue color for tank
            red_reg <= 4'h0;
            green_reg <= 4'h0;
            blue_reg <= 4'hA;
         end
         else if (bullet_pixel_on) begin
            // White color for bullets
            red_reg <= 4'hF;
            green_reg <= 4'hF;
            blue_reg <= 4'hF;
         end
         else if (score_display_on && score_display_pixel) begin
            // White color for score text
            red_reg <= 4'hF;
            green_reg <= 4'hF;
            blue_reg <= 4'hF;
         end
         else begin
            // Black background
            red_reg <= 4'h0;
            green_reg <= 4'h0;
            blue_reg <= 4'h0;
         end
      end
```

```
3'b010: begin  // Level 1 Win Screen
        h_sync_reg <= win1_h_sync;
        v_sync_reg <= win1_v_sync;
        red_reg <= win1_red;
        green_reg <= win1_green;
        blue_reg <= win1_blue;
    end

3'b011: begin  // PLAYING_LEVEL2 state
    if (alien_pixel_on) begin
        // Green color for Level 2 aliens
        red_reg <= 4'h0;
        green_reg <= 4'hB;
        blue_reg <= 4'h0;
    end
    else if (tank_pixel_on) begin
        // Dark blue color for tank
        red_reg <= 4'h0;
        green_reg <= 4'h0;
        blue_reg <= 4'hA;
    end
    else if (bullet_pixel_on) begin
        // White color for bullets
        red_reg <= 4'hF;
        green_reg <= 4'hF;
        blue_reg <= 4'hF;
    end
    else if (score_display_on && score_display_pixel) begin
        // White color for score text
        red_reg <= 4'hF;
        green_reg <= 4'hF;
        blue_reg <= 4'hF;
    end
    else begin
        // Black background
        red_reg <= 4'h0;
        green_reg <= 4'h0;
        blue_reg <= 4'h0;
    end
end

3'b100: begin  // Lose Screen
```

```
                h_sync_reg <= lose_h_sync;
                v_sync_reg <= lose_v_sync;
                red_reg <= lose_red;
                green_reg <= lose_green;
                blue_reg <= lose_blue;
            end

          3'b101: begin  // LEVEL 2 WIN state
                h_sync_reg <= win2_h_sync;
                v_sync_reg <= win2_v_sync;
                red_reg <= win2_red;
                green_reg <= win2_green;
                blue_reg <= win2_blue;
            end

           default: begin  // Fallback to Start Screen
                h_sync_reg <= start_h_sync;
                v_sync_reg <= start_v_sync;
                red_reg <= start_red;
                green_reg <= start_green;
                blue_reg <= start_blue;
            end
          endcase
    end
end
    // Assign RGB outputs
    assign red = red_reg;
    assign green = green_reg;
    assign blue = blue_reg;

endmodule
```

2. The Keyboard Module which tells us which key has been pressed:

```
`timescale 1ns / 1ps

module keyboard_1 (
    input wire clk, reset,
    input wire ps2d, ps2c,
    output SPACE,
```

```
    output UP,
    output DOWN,
    output LEFT,
    output RIGHT,
    output PAUSE,
    output RESET,
    output SPECIAL,
    output reg state,
    output reg key_release
);

// Declare variables
wire [7:0] dout;
wire rx_done_tick;
wire rx_en = 1'b1; // Always enable receiver

// Instantiate PS/2 receiver
ps2_rx ps2_receiver (
    .clk(clk),
    .reset(reset),
    .ps2d(ps2d),
    .ps2c(ps2c),
    .rx_en(rx_en),
    .rx_done_tick(rx_done_tick),
    .dout(dout)
);

// Sequential logic
reg [7:0] key;
always @(posedge clk, posedge reset)
begin
    if (reset) begin
        key <= 16'h00;
        key_release <= 1'b0;
        state <= 1'b0;
    end
    else if (rx_done_tick)
    begin
        key <= dout; // key is one rx cycle behind dout
        if (key == 16'hf0) // check if the key has been released
            key_release <= 1'b1;
        else
            key_release <= 1'b0;
```

```
    end
end


// Key Assignments
assign SPACE = (key == 16'h29) & !key_release; // Space Bar
assign UP = (key == 16'h1D) & !key_release;    // W
assign DOWN = (key == 16'h1B) & !key_release;  // S
assign LEFT = (key == 16'h1C) & !key_release;  // A
assign RIGHT = (key == 16'h23) & !key_release; // D
assign PAUSE = (key == 16'h4D) & !key_release; // P
assign RESET = (key == 16'h2D) & !key_release; // R
assign SPECIAL = (key == 16'h22) & !key_release; // X

// State change logic
always @(posedge clk) begin
    if (rx_done_tick) begin
        case(key)
            16'h5A : state <= 1'b1; // Enter key
            16'h66 : state <= 1'b0; // Backspace key
            default: state <= state;
        endcase
    end
end
endmodule
```

```
module ps2_rx(
    input wire clk, reset,
    input wire ps2d, ps2c, rx_en,
    output reg rx_done_tick,
    output wire [7:0] dout
);

// Symbolic state declaration
localparam [1:0]
    idle = 2'b00,
    dps  = 2'b01, // data, parity, stop
    load = 2'b10;

// Signal declaration
reg [1:0] state_reg, state_next;
```

```
reg [7:0] filter_reg;
wire [7:0] filter_next;
reg f_ps2c_reg;
wire f_ps2c_next;
reg [3:0] n_reg, n_next;
reg [10:0] b_reg, b_next;
wire fall_edge;


// Filter and falling-edge tick generation for ps2c
always @(posedge clk, posedge reset)
if (reset)
begin
   filter_reg <= 0;
   f_ps2c_reg <= 0;
end
else
begin
   filter_reg <= filter_next;
   f_ps2c_reg <= f_ps2c_next;
end

assign filter_next = {ps2c, filter_reg[7:1]};
assign f_ps2c_next = (filter_reg==8'b11111111) ? 1'b1 :
                     (filter_reg==8'b00000000) ? 1'b0 :
                     f_ps2c_reg;
assign fall_edge = f_ps2c_reg & ~f_ps2c_next;

// FSMD state & data registers
always @(posedge clk, posedge reset)
if (reset)
begin
   state_reg <= idle;
   n_reg <= 0;
   b_reg <= 0;
end
else
begin
   state_reg <= state_next;
   n_reg <= n_next;
   b_reg <= b_next;
end
```

```
// FSMD next-state logic
always @*
begin
   state_next = state_reg;
   rx_done_tick = 1'b0;
   n_next = n_reg;
   b_next = b_reg;
   case (state_reg)
      idle:
         if (fall_edge & rx_en)
         begin
            // Shift in start bit
            b_next = {ps2d, b_reg[10:1]};
            n_next = 4'b1001;
            state_next = dps;
         end
      dps: // 8 data + 1 parity + 1 stop
         if (fall_edge)
         begin
            b_next = {ps2d, b_reg[10:1]};
            if (n_reg==0)
               state_next = load;
            else
               n_next = n_reg - 1;
         end
      load: // 1 extra clock to complete the last shift
         begin
            state_next = idle;
            rx_done_tick = 1'b1;
         end

      default: begin
         // Safe fallback for unhandled cases
      end
   endcase

end

// Output
assign dout = b_reg[8:1]; // Data bits

endmodule
```

3. Python function to convert PNG to a bitmap of desired dimensions:

```
import numpy as np
from PIL import Image

# 512 x 256 for start screen
# 256 x 128 for other screens

def image_to_bitmap(image_path, output_file, width=512, height=256):
    img = Image.open(image_path).convert('L')  # Convert to grayscale

    img_resized = img.resize((width, height), Image.Resampling.LANCZOS)

    binary_img = np.array(img_resized) < 128

    with open(output_file, 'w') as f:
        for y in range(height):
            binary_row = ''.join(['1' if binary_img[y, x] else '0' for x in range(width)])
            f.write(f"main[{y}] = {width}'b{binary_row};\n")
    print(f"Conversion complete. Output size: {height} rows x {width} columns.")
    print(len(binary_row))

# image_to_bitmap('game_over.png', 'sprite_bitmap.txt')

# image_to_bitmap('Level1_Win.png', 'sprite_bitmap.txt')

# image_to_bitmap('Level2_Win.png', 'sprite_bitmap.txt')

image_to_bitmap('start_screen.png', 'sprite_bitmap.txt')
```

6. Challenges

- Initially, we attempted to use the joystick as the input peripheral instead of the keyboard, and at first we tried to connect the joystick directly to the FPGA but it didn't work as expected. After referring to the joystick documentation provided by the instructor, we decided to connect the joystick to the FPGA through a breadboard.

  Upon checking the voltages, we observed that the Y-axis was mapped incorrectly. Specifically, the voltage decreased when moving in the positive Y direction and

increased in the negative Y direction, whereas it should have behaved oppositely. We made changes in our code accordingly but did not get the desired outputs.

We used the LEDs on the FPGA to display the joystick movement output. While the push button output worked perfectly, the LEDs fluctuated when moving the joystick along the X or Y axis.

After much consideration, we decided to shift to a keyboard and have successfully integrated the game with the keyboard inputs.

- Figuring out how to handle the level transition from level 1 to level 2 also provided a significant challenge as we had to revise our game state FSM logic for this many times to get it to work. After that, we saw that there was a problem with the way we're handling reset condition in the aliens_movement modules so we had to add another explicit level_reset wire to trigger the reset state according to the game state FSM logic.

7. References

1. GitHub - Lucario2319/keyboard_module_guide
2. https://github.com/klam20/FPGAProjects/tree/main/VGATextGeneration

8. Github link for code:

https://github.com/amnnaqvi/Disaster-Strikes-Digital-Logic-Design (SpaceForceX folder)