MIDDLE EAST TECHNICAL UNIVERSITY
DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

EE314
DIGITAL ELECTRONICS LABORATORY
SPRING ‘24

TERM PROJECT REPORT
FPGA IMPLEMENTATION OF ISOMETRIC SHOOTER GAME

GROUP 50

Batuhan Elmas	2516086
Basri Kaan Demirkoparan	2442903
Sümeyra Arıcan	2442432
Zülal Uludoğan	2444057

I. INTRODUCTION

This report examines the term project for the Digital Electronics Laboratory course, where the primary task is to create an isometric shooter game using Verilog HDL on an FPGA platform. The report begins with a detailed problem definition, outlining the game's objectives and specifications. It then provides an overview of the VGA interface used for game display, followed by an in-depth explanation of the solution approach, including the design and implementation of key game components. The challenges faced during development and their proposed solutions are discussed, followed by an evaluation of the results, highlighting how the project meets the specified criteria. Finally, the report concludes with a general discussion for the project, its contributions and future developments.

II. PROBLEM DEFINITION

The objective of this project is the development of the game's logic, visual interface, and the interaction mechanisms via the FPGA hardware. The game is inspired by classic arcade shooters, specifically taking cues from the iconic Space Invaders game. Specifications given in the project description file are as follows, the player controls a central spaceship situated in the middle of a game field. This spaceship can rotate but cannot move laterally and must defend against enemies that appear at the boundaries and move towards the center. The player must strategically rotate the spaceship to aim and fire projectiles, destroying the incoming enemies before they reach and collide with the spaceship, which would end the game. The game field will be displayed using a VGA interface, supporting a resolution of 640 x 480 pixels. The enemies will spawn at predefined angles, move towards the spaceship, and vary in type and health. The player will have two shooting modes to choose from, offering different projectile spreads and damage levels.

III. VIDEO GRAPHICS ARRAY (VGA)

The Video Graphics Array (VGA) interface is a critical component in this project, since it serves as the medium through which the game field and various visual elements are rendered on a display. This section will summarize the background knowledge gained on the VGA interface while creating the project.

VGA (Video Graphics Array) is a standard for analog video signals, comprising RGB color signals and synchronization signals HSYNC and VSYNC. To guarantee correct video display, signal creation requires precise timing, which includes synchronization pulses and pixel clocks. For instance, exact timings for active video, front porch, sync pulse, and back porch are observed at 640x480 resolution and 60Hz refresh rate. This temporal logic can be implemented, and the required VGA signals can be produced using hardware such as FPGAs [1]. A thorough understanding of VGA standards, memory management, state-driven design, and practical coding techniques was required for the VGA project to create a game rendering module using Verilog. Supporting a 640x480 resolution, producing accurate timing signals, controlling color depth, and maintaining a frame buffer for pixel data were among the essential needs. Using specified coordinates and dynamic updating based on game status, the project needed to effectively store and retrieve picture data for various game elements, including the background, ship, enemies, score, and game-over message. We learned how to create synchronization signals for VGA, the significance of precise timing, and the benefits of modular design for more straightforward maintenance and debugging throughout the project. State machines were essential for maintaining various game states and rendering matching elements. We learned how to use Verilog to handle massive data volumes, guarantee effective real-time performance, and use debugging tools to confirm the design before hardware implementation. [2]

At first, we worked on obtaining white, blue, and purple screens via VGA. After achieving this, we aimed to print a still image on the screen in our case a simple smile face. Then, we tried moving an image over VGA by following a tutorial we found. After doing all those trials and getting used to vga control we started implementing our needs for our project step by step. After successful implementation, we tried to make the project more efficient by adding more features. Finally, we were successful in creating a working VGA driver for our project.

IV. SOLUTION APPROACH

This section outlines the proposed solution for developing the game. Firstly, the overall system architecture is illustrated using a block diagram, which highlights the communication and interaction between the submodules such as spaceship control, enemy dynamics, and the VGA interface. Then, each submodule is described in detail, including the design decisions, implementation strategies, and the rationale behind choosing specific methods or algorithms. Diagrams, state diagrams, and pseudocodes are also provided where necessary to illustrate better.

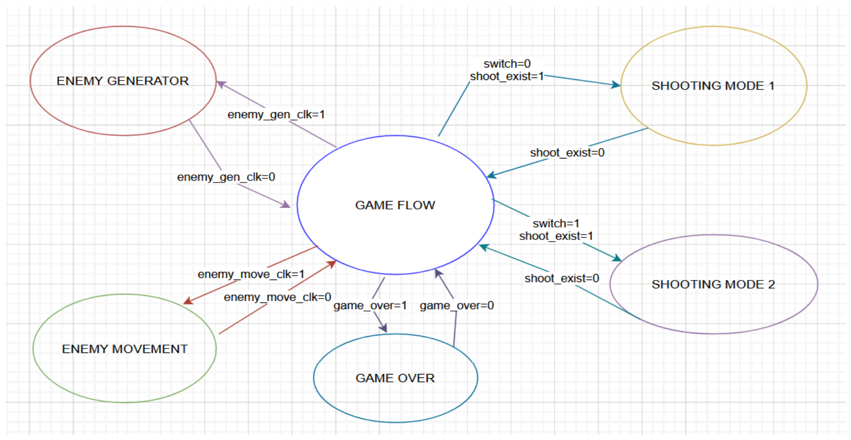
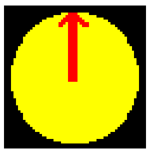


Figure 1. Block Diagram

Spaceship Control

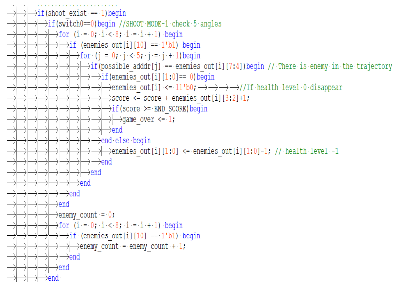


Although we first considered using an actual spaceship image for our ship, we realized that pixel distortion increased significantly when rotating it at sixteen different angles. Instead of fixing pixel distortions, we changed our shape to use time more effectively. We chose a round, yellow shape with a red arrow to indicate the angle.

Figure 2. Spaceship Image

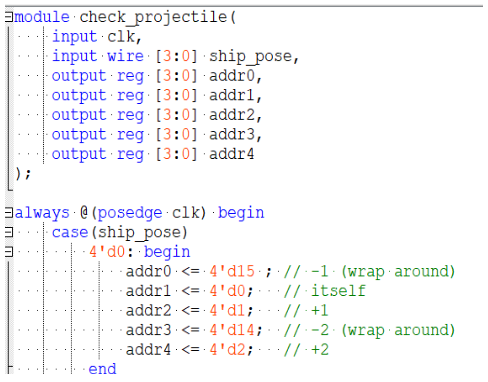
After that, we redrew the shape at sixteen different angles to provide the rotation look. The project description required the spaceship to rotate clockwise and counterclockwise. We defined two different buttons on the FPGA to provide two different rotation directions. We assigned KEY[0] as ccw and KEY[1] as cw.

The input clock (clk) was divided down to a slower signal (clk_out) using a clock divider to manage button presses effectively without debouncing issues. Updates to the ship's position, stored in a 4-bit register (ship_pose), occur on the positive edge of clk_out. Pressing the counterclockwise button increments ship_pose, wrapping around from 15 (1111) back to 0 (0000). Pressing the clockwise button decrements ship_pose, wrapping around from 0 to 15. The reset signal initializes ship_pose to 0 and resets the clock divider, ensuring the system begins in a defined state.



```
if(shoot_exist == 1)begin
    if(switch0==0)begin //SHOOT MODE-1 check 5 angles
        for (i = 0; i < 8; i = i + 1) begin
            if (enemies_out[i][10] == 1'b1) begin
                for (j = 0; j < 5; j = j + 1)begin
                    if(possible_adddr[j] == enemies_out[i][7:4])begin // There is enemy in the trajectory
                        if(enemies_out[i][1:0]== 0)begin
                            enemies_out[i] <= 11'b0;                //If health level 0 disappear
                            score <= score + enemies_out[i][3:2]+1;
                            if(score >= END_SCORE)begin
                                game_over <= 1;
                            end
                        end else begin
                            enemies_out[i][1:0] <= enemies_out[i][1:0]-1; // health level -1
                        end
                    end
                end
            end
        end
        enemy_count = 0;
        for (i = 0; i < 8; i = i + 1) begin
            if (enemies_out[i][10] -- 1'b1) begin
                enemy_count = enemy_count + 1;
            end
        end
    end
```

Shooting Modes: We implemented two shooting modes, one mode fires projectiles along three trajectories(45° angles), while the other mode uses five trajectories(90° angles). We toggled between these modes using switch SW[0]. The selected shooting mode is visually indicated in the lower right corner of the screen, displaying the number 1 for five and 2 for three trajectories. Code for shooting mode 1 can be seen in Figure 3.(Check Appendix for Figure 3). *Figure 3. Pseudo Code for Shooting Mode 1*



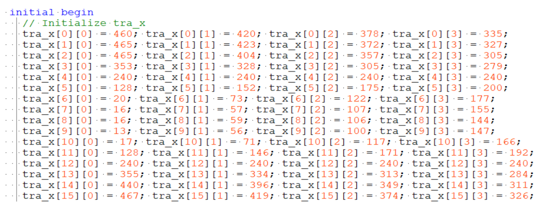
```
module check_projectile(
    input clk,
    input wire [3:0] ship_pose,
    output reg [3:0] addr0,
    output reg [3:0] addr1,
    output reg [3:0] addr2,
    output reg [3:0] addr3,
    output reg [3:0] addr4
);

always @(posedge clk) begin
    case(ship_pose)
        4'd0: begin
            addr0 <= 4'd15 ; // -1 (wrap around)
            addr1 <= 4'd0;   // itself
            addr2 <= 4'd1;   // +1
            addr3 <= 4'd14;  // -2 (wrap around)
            addr4 <= 4'd2;   // +2
        end
```

The spaceship's location was crucial for determining the shooting direction, so we wrote a Verilog module, `check_projectile`, to handle this. This module updates five 4-bit address outputs (`addr0` to `addr4`) based on the current ship position (`ship_pose`). On each clock signal's positive edge, it checks `ship_pose` and assigns addresses representing the ship's current and adjacent positions. For instance, if `ship_pose` is `0`, `addr0` is `15` (wrap-around for -1), `addr1` is `0`, `addr2` is `1`, `addr3` is `14` (wrap-around for -2), and `addr4` is `2`. This ensures each address output accurately reflects the ship's position and its surroundings.

Figure 4. Pseudo Code for Checking Projectile

Then, when the shoot occurs (when the key is pressed), we wanted this action to be sent as input to the game engine, so we wrote a module to ensure this. This Verilog module `shoot_detector` synchronizes an input signal to the clock, detects negative edges on this synchronized signal, and outputs `shoot_exist` to indicate when a shooting event has occurred. Finally, we wrote a code snippet that checks if "shoot_exist" is "1". If true, it indicates that an attraction event has entered a conditional block. Inside this block, SW0 is also controlled, which controls the mode of the game. Depending on the Shooting mode, it tries to shoot enemies in three or five different trajectories.

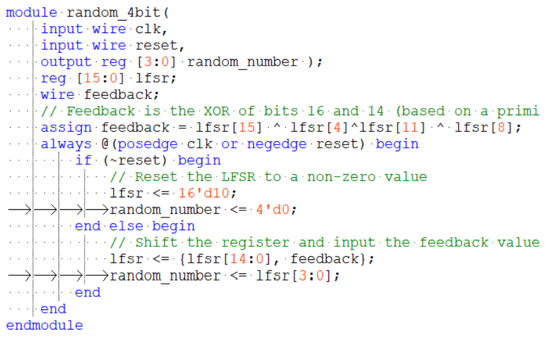
Enemy Dynamics



```
initial begin
    // Initialize tra_x
    tra_x[0][0] = 460; tra_x[0][1] = 420; tra_x[0][2] = 378; tra_x[0][3] = 335;
    tra_x[1][0] = 465; tra_x[1][1] = 423; tra_x[1][2] = 372; tra_x[1][3] = 327;
    tra_x[2][0] = 465; tra_x[2][1] = 404; tra_x[2][2] = 357; tra_x[2][3] = 305;
    tra_x[3][0] = 353; tra_x[3][1] = 328; tra_x[3][2] = 305; tra_x[3][3] = 279;
    tra_x[4][0] = 240; tra_x[4][1] = 240; tra_x[4][2] = 240; tra_x[4][3] = 240;
    tra_x[5][0] = 128; tra_x[5][1] = 152; tra_x[5][2] = 175; tra_x[5][3] = 200;
    tra_x[6][0] = 20; tra_x[6][1] = 73; tra_x[6][2] = 122; tra_x[6][3] = 177;
    tra_x[7][0] = 16; tra_x[7][1] = 57; tra_x[7][2] = 107; tra_x[7][3] = 155;
    tra_x[8][0] = 16; tra_x[8][1] = 59; tra_x[8][2] = 106; tra_x[8][3] = 144;
    tra_x[9][0] = 13; tra_x[9][1] = 56; tra_x[9][2] = 100; tra_x[9][3] = 147;
    tra_x[10][0] = 17; tra_x[10][1] = 71; tra_x[10][2] = 117; tra_x[10][3] = 166;
    tra_x[11][0] = 128; tra_x[11][1] = 146; tra_x[11][2] = 171; tra_x[11][3] = 192;
    tra_x[12][0] = 240; tra_x[12][1] = 240; tra_x[12][2] = 240; tra_x[12][3] = 240;
    tra_x[13][0] = 355; tra_x[13][1] = 334; tra_x[13][2] = 313; tra_x[13][3] = 284;
    tra_x[14][0] = 440; tra_x[14][1] = 396; tra_x[14][2] = 349; tra_x[14][3] = 311;
    tra_x[15][0] = 467; tra_x[15][1] = 419; tra_x[15][2] = 374; tra_x[15][3] = 326;
```

We designed a background displaying sixteen angles to easily determine enemy arrival coordinates, simplifying calculations for challenging angles like 22.5°. To ensure enemies move every clock cycle, we listed suitable pixel values from the background image and integrated these into our VGA code, programming enemies to consistently move from the outside towards the center.

Figure 5. Pseudo Code for Possible X Coordinates for Enemy (Check Appendix for Figure 5).



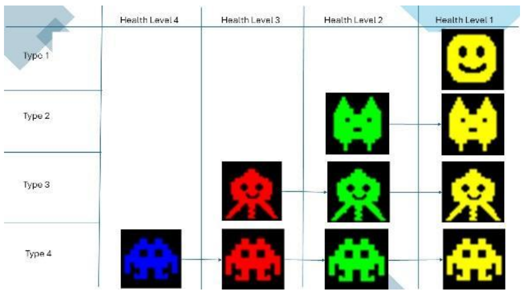
```
module random_4bit(
    input wire clk,
    input wire reset,
    output reg [3:0] random_number );
    reg [15:0] lfsr;
    wire feedback;
    // Feedback is the XOR of bits 16 and 14 (based on a primi
    assign feedback = lfsr[15] ^ lfsr[4]^lfsr[11] ^ lfsr[8];
    always @(posedge clk or negedge reset) begin
        if (~reset) begin
            // Reset the LFSR to a non-zero value
            lfsr <= 16'd10;
            random_number <= 4'd0;
        end else begin
            // Shift the register and input the feedback value
            lfsr <= {lfsr[14:0], feedback};
            random_number <= lfsr[3:0];
        end
    end
endmodule
```

In the project, some randomness was requested from us regarding the enemy. We were familiar with the problems in Verilog generating pseudorandom number section from the lab sessions during the period. So, we decided to use the technique taught in the laboratory. We needed two separate randomness values in our project—a 2-bit random number for the enemy type and a 4-bit random number for the enemy's angle of incidence. Therefore, we used two separate LFSRs of 2-bit and 4-bit.

Figure 6. Pseudo Code for 4-bit LFSR

To maintain a balanced enemy population in our game, we implemented two strategies based on the current number of enemies. When the count is between three and seven, we spawn one enemy periodically using `enemy_gen_clk`. If there are fewer than three enemies, we immediately spawn two new enemies using separate random generators. This approach prevents overcrowding and ensures there are always between two and eight enemies on screen. These mechanisms optimize gameplay dynamics by balancing enemy presence and avoiding underrepresentation or overlapping enemies.

In our game, enemy spawning is synchronized with the clock, randomly generating a number of enemies at random addresses every four seconds. We chose the clock frequency based on the time required for player actions like turning and shooting the spaceship. Additionally, we implemented two game modes: easy and hard. In hard mode, the enemy spawn speed is increased to provide a greater challenge.



We drew different enemies with different shapes and health levels using pixel drawing. Blue indicates four health levels, red indicates three health levels, green indicates two health levels, and yellow indicates one health level. The health level is expressed in [1:0] bits of the enemy, each expressed in eleven bits. Enemies' health count should decrease every time they are hit with a projectile, and we achieved this by printing the enemy image with less health (different color) in VGA when a successful shooting occurs.

Figure 7. Enemy Types and Health Levels

In the game engine's main code, we have generated eight regs for each enemy to store their info. enemy_out[9:4] stores the enemy address. The first 2 bits show the distance of the enemy to the spaceship. Moreover, the other 4 bits show the trajectory of the enemy. In every enemy_move_clk, The first 2 bit adds with 1. However, the trajectory of the enemy does not change.

Player Score and Game Over Conditions : In the scoring system, shooting mode and events are checked. In "SHOOT MODE-1," the game iterates over up to 8 active enemies, checking five possible trajectories per enemy for projectile interference. If a match is found, the enemy's health is reduced by one. When an enemy's health reaches zero, it is removed, and the player's score increases by the enemy's initial health value. If the score reaches or exceeds 20, the game is won. "SHOOT MODE-2" checks three trajectories per enemy, but otherwise follows the same process. The number of remaining active enemies is updated accordingly. This system effectively manages shooting mechanics, enemy health, scoring, and game progression.

- Game Over : As we mentioned before, we determined possible positions for the enemies. We kept these positions constant, with four at each trajectory, sixty-four in total. In other words, when an enemy appears on the screen, it can move a maximum of 4 steps. When it reached step five, we accepted it as a collision with the spaceship and programmed the game over the situation this way. When we detect game over condition, the text “game over” is printed on the screen.

Coding Approach

We firstly determined the inputs and outputs of the game engine code.

INPUTS : input wire clk, input wire reset, input wire button_shoot, input wire button_cw, input wire button_ccw, input wire switch0, input wire switch1,

OUTPUTS : output wire [3:0] ship_pose, output reg [10:0] enemy0_out, output reg [10:0] enemy1_out, output reg [10:0] enemy2_out, output reg [10:0] enemy3_out, output reg [10:0] enemy4_out, output reg [10:0] enemy5_out, output reg [10:0] enemy6_out, output reg [10:0] enemy7_out, output reg [4:0] score, output reg shoot_mode, output reg game_over

Code Hierarchy

➔ Image (Top Module)
- Main_Activity (Combines sub modules and directs output to top module)
    - VGAInterface (combines the addresses of each pixel, generate Hsynch-Vsynch signals and outputs the corresponding RGB output to VGA)
        1. Counter (simple counter used inside PixCounter)
        2. PixCounter (detects an image)
    - Picture (Reads enemy images, spaceship images etc. , stores them in the memory, attaches pixel addresses to corresponding pictures)
        1. PositionDecoder (Enemy movement)

2. ScoreBoard (Score)

- Game Engine (Game architecture)
  - Random_2bit (2-bit LFSR)
  - Random_4bit (4-bit LFSR)
  - Shoot_Detector (Checks whether shoot)
  - Ship_Controller (Spaceship position)
  - Check_Projectile (Positions where the ship can projectile)
- Score Board Display (Prints score on VGA)
  - ScoreBoard (Score)
  - sscm (Seven Segment Display)

Game_engine code outputs includes everything we want to display on the monitor. VGA modules takes these outputs in every clock. So when the outputs of the game_engine code updated, the images are also updated.

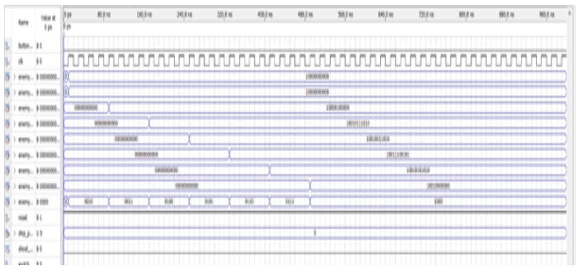
V. CHALLENGES

This section addresses the various challenges encountered during the development of the game. Each challenge is discussed in detail, along with the strategies and solutions implemented to overcome them.

Our first challenge was printing images to VGA. None of the group members had any experience with VGA before, and it took time to learn how to do it because there were limited resources about VGA, especially on the internet. As a result, we converted the images in PNG format to list format using MATLAB code. We expressed the colors as 8-bit hexadecimal numbers[3]. The second challenge we faced was to adjust the trajectories that enemies could come. We overcame this difficulty by listing the x-y pixel numbers of suitable positions for enemies and using them in appropriate places. We obtained the x-y coordinates from the background image. In this way, it has become easier to calculate angles that are difficult to obtain, such as 22.5 degrees. The third challenge was storing enemy information for up to eight enemies on the screen with some random attributes. We chose to use an 11-bit array for simplicity, storing health indications and enemy positions efficiently. The array allocated 1 bit for the enemy's existence, 6 bits for the address, 2 bits for the enemy type, and 2 bits for the enemy health level. We derived the 4-bit angle part of the address from a 4-bit LFSR and the enemy type from a 2-bit LFSR. For visual appeal, we used custom pixel drawings instead of basic shapes like squares and circles.

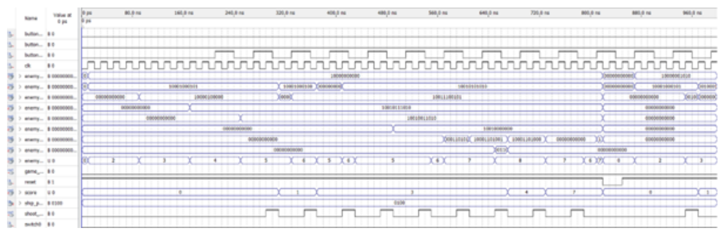
VI. RESULTS

This section discusses the outcomes of the project, evaluating how effectively the implementation meets the defined objectives and requirements. It includes an assessment of game functionality, performance metrics, and visual output quality. Additionally, this section provides a comparison of the final product against the initial goals and it highlights the strong and weak parts of the final implementation.



Since we did not have access to the screen or FPGA at all times, or because compiling took time, we proceeded by checking the operation of each game engine module through simulation. Figure X shows the simulation results of random enemy generation. Since we also used 2-bit and 4-bit LFSRs in this module, we were sure many modules were working correctly, thanks to the simulation result (Figure 8) (Check Appendix for Figure 8).

Figure 8. Simulation Results for random enemy generation



The only game-over condition of the game was for the enemy to collide with the spaceship. Since it was tough to test via VGA in the early stages, testing via simulation (Figure 9) (Check Appendix for Figure 9). saved us much time and allowed us to promptly fix errors in the code.

Figure 9. Simulation Results for game-over condition

We frequently used simulations to check for code corruption and debug new modules added to the game engine. This step-by-step approach, verifying each part with simulations, was crucial for achieving an error-free result. Early and quick debugging prevented complications and time waste as the code grew more complex, allowing us to develop a good game within a limited timeframe.

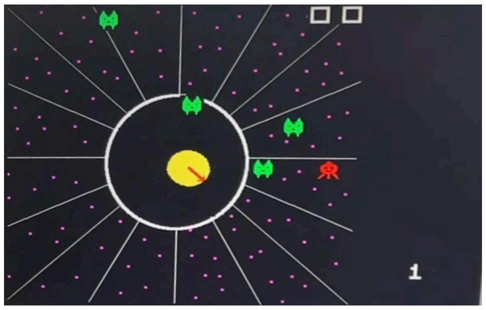


Figure 10 contains an image of our game screen. On this screen, it can be seen the enemies from the random trajectories, the scoreboard at the top right, the spaceship in the center that is easily understood in which direction it is turning, the background image showing possible trajectories, and the number at the bottom right showing which shooting mode is selected. Although visual output quality can be further improved, pixel distortions are negligible. The game's speed is adjusted by considering the time needed to press the keys.

Figure 10. Game screen

Despite challenges, our game successfully implements its rules with minor issues. Enemies' paths, types, and health levels are randomized and can be eliminated effectively. The scoreboard accurately updates, adding enemy health to the score upon their defeat. At game over, the score and restart option are displayed. Additional features include a background image, a seven-segment score display, and multiple game levels. However, the game's visual aspects need improvement to align with our initial goals. For instance, while detecting enemy hits is accurate, the lack of visible projectiles on screen detracts from the overall visual experience.

VII. CONCLUSION

In this FPGA-based Space Invaders game project, we designed a dynamic gameplay environment with a central player's spaceship. Players control the spaceship's rotation using FPGA buttons for strategic positioning and aiming. Enemies appear randomly at specific angles (e.g., 0°, 22.5°, 45°) to add unpredictability. The number of enemies varies between 2 and 8 to maintain a consistent challenge. Different enemy types with varying health levels and shapes add complexity. Enemies move radially toward the spaceship. The game offers two shooting modes: Mode 1 (wider spray) and Mode 2 (narrower spray), switchable via FPGA switches. Players can also select easy or hard mode using a switch. Visual indicators display the damage inflicted on enemies and their remaining health, providing feedback to the player. Despite the foundational knowledge from EE348 and EE314 on digital design and FPGA implementation, we learned many new concepts from scratch for this project, particularly in Verilog and VGA, through research. While challenging, it was often enjoyable. Creating a game within a limited timeframe also enhanced our skills in time management and teamwork. If we had more time for this project, we could have put effort into the visual aesthetics. For example, we would choose a more complex image for the spaceship and add animation to the enemies. We could also add more details to the leveling system.

VIII. REFERENCES

[1] "VGA Microcontroller projects". TinyVGA.com. http://tinyvga.com/vga-timing

[2] Teras IC. (2015, April 2). DE1-SOC User Manual [Online]. Available: http://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=836&FID=3a3708b0790bb9c721f94909c5ac96d6

[3] Klumbys, M. (2017) *Image from FPGA to VGA*, *Instructables*. Available at: https://www.instructables.com/Image-from-FPGA-to-VGA/

IX. APPENDIX / APPENDICES

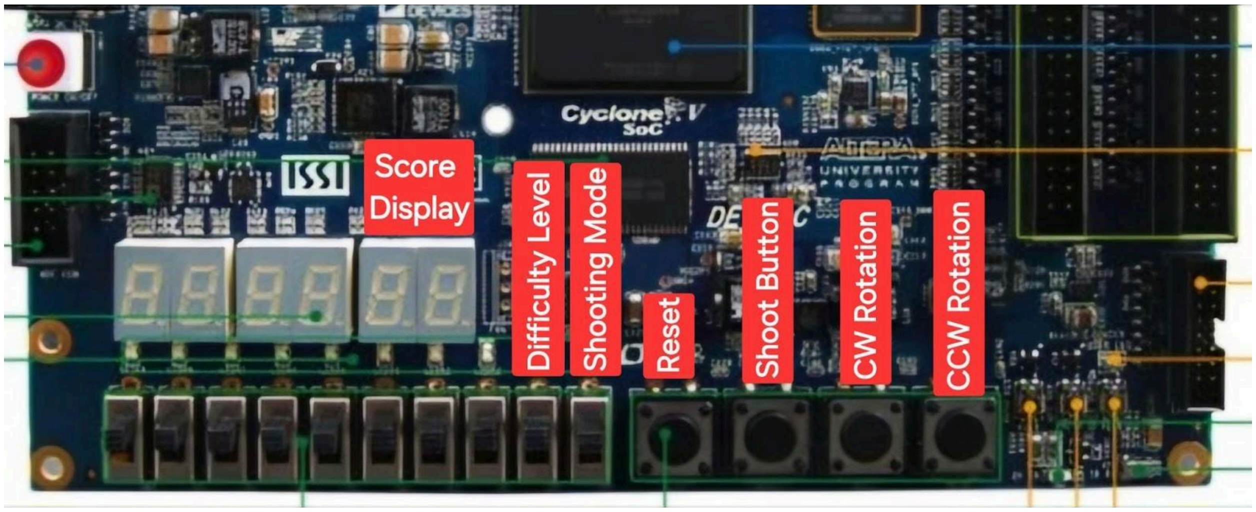


Figure 11. FPGA Demonstration

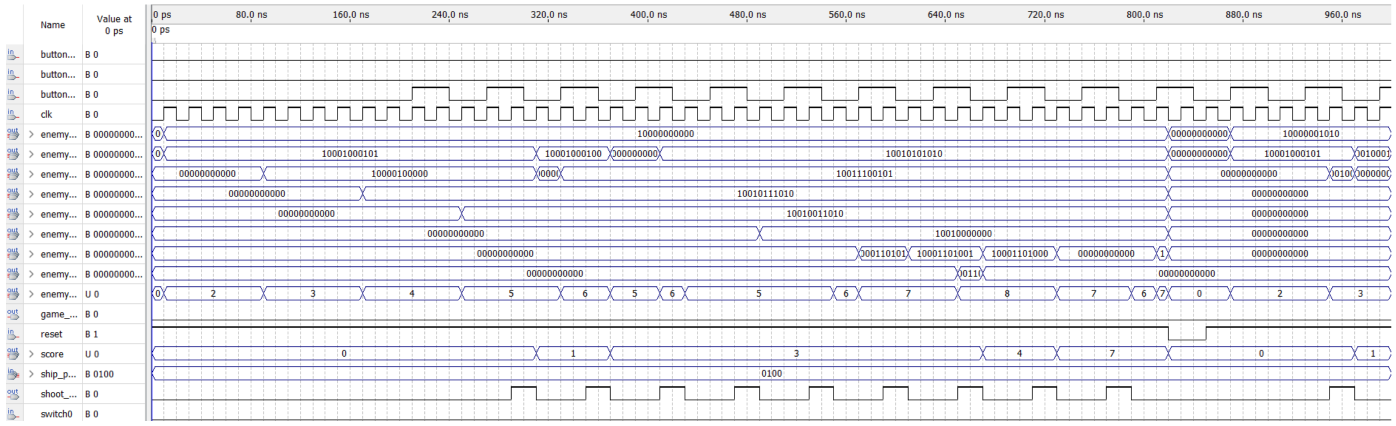


Figure 12. *Zoomed in version of Figure 10.*

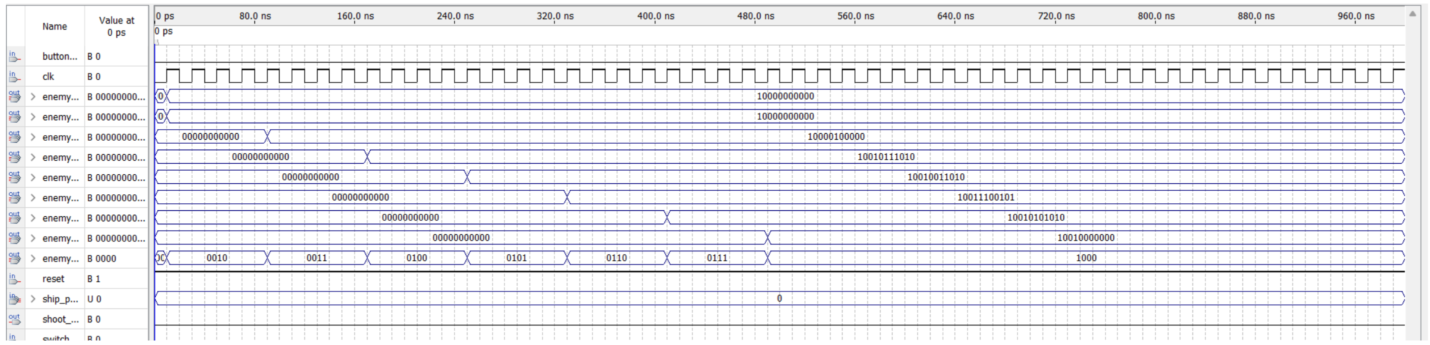


Figure 13. *Zoomed in version of Figure 8.*

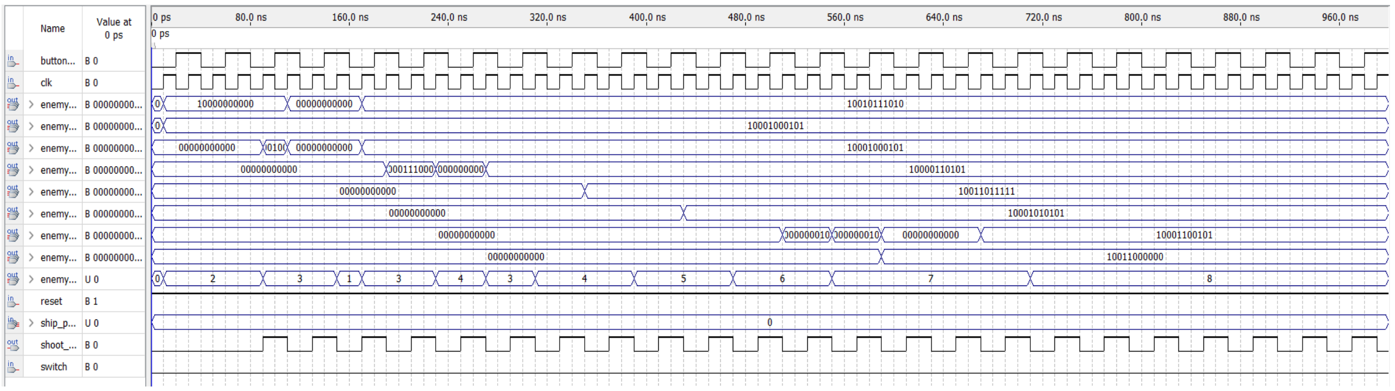

Name	Value at 0 ps
button...	B 0
clk	B 0
enemy...	B 00000000...
enemy...	B 00000000...
enemy...	B 00000000...
enemy...	B 00000000...
enemy...	B 00000000...
enemy...	B 00000000...
enemy...	B 00000000...
enemy...	B 00000000...
enemy...	U 0
reset	B 1
ship_p...	U 0
shoot_...	B 0
switch	B 0

Figure 14. Simulation Results for production of new enemies.

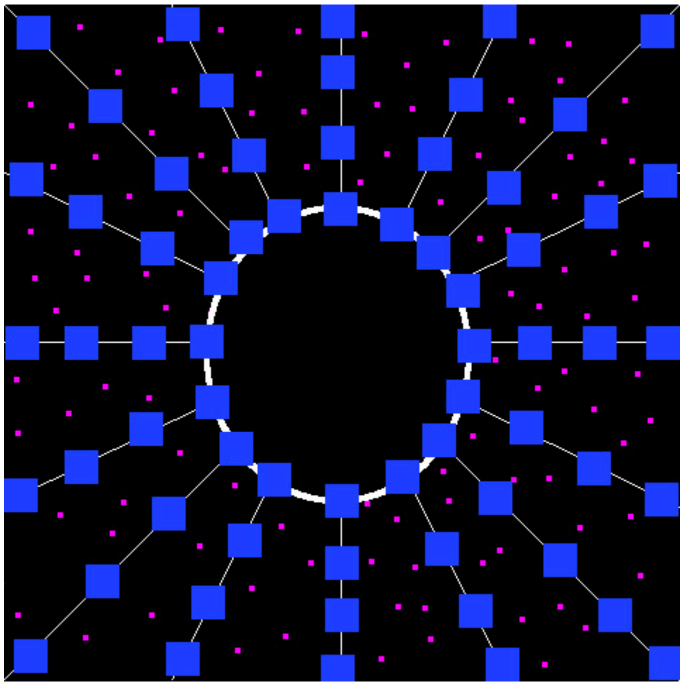


Figure 15. Possible Enemy Positions

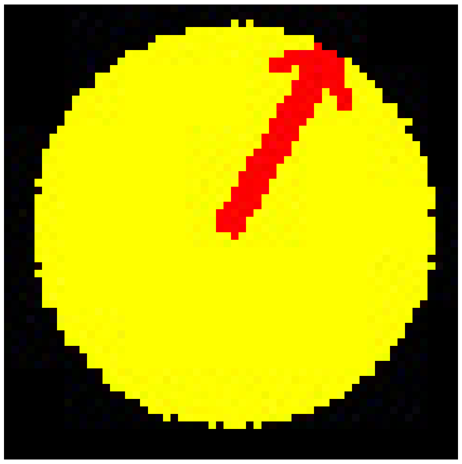
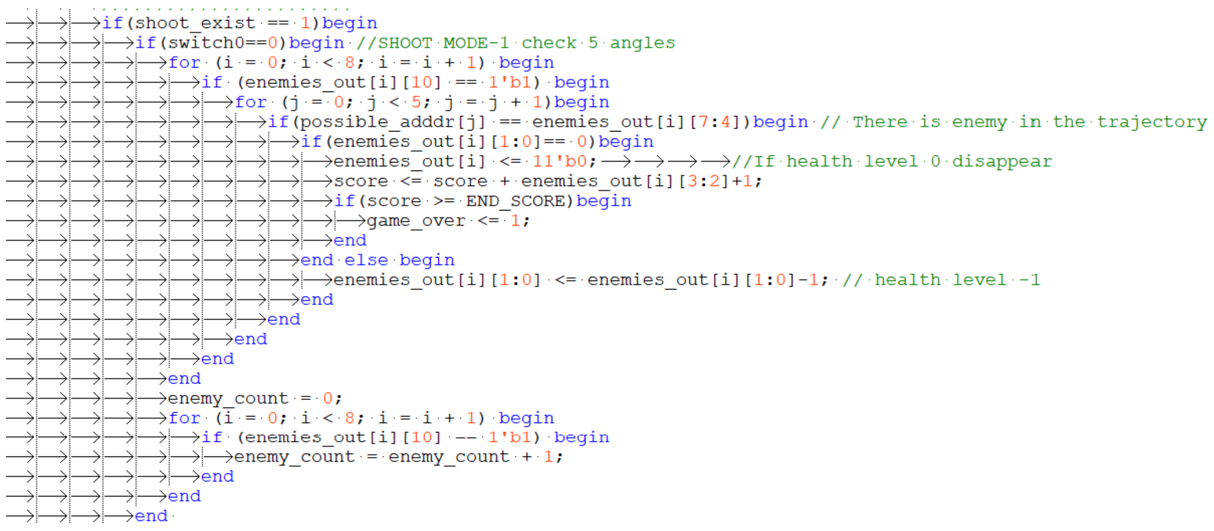


Figure 16. Different spaceship position (22.5 degrees)



```
if(shoot_exist == 1)begin
    if(switch0==0)begin //SHOOT MODE-1 check 5 angles
        for (i = 0; i < 8; i = i + 1) begin
            if (enemies_out[i][10] == 1'b1) begin
                for (j = 0; j < 5; j = j + 1)begin
                    if(possible_adddr[j] == enemies_out[i][7:4])begin // There is enemy in the trajectory
                        if(enemies_out[i][1:0]== 0)begin
                            enemies_out[i] <= 11'b0;                //If health level 0 disappear
                            score <= score + enemies_out[i][3:2]+1;
                            if(score >= END_SCORE)begin
                                game_over <= 1;
                            end
                        end else begin
                            enemies_out[i][1:0] <= enemies_out[i][1:0]-1; // health level -1
                        end
                    end
                end
            end
        end
    end
    enemy_count = 0;
    for (i = 0; i < 8; i = i + 1) begin
        if (enemies_out[i][10] -- 1'b1) begin
            enemy_count = enemy_count + 1;
        end
    end
end
```

Figure 17.. Zoomed in version of Figure 3.

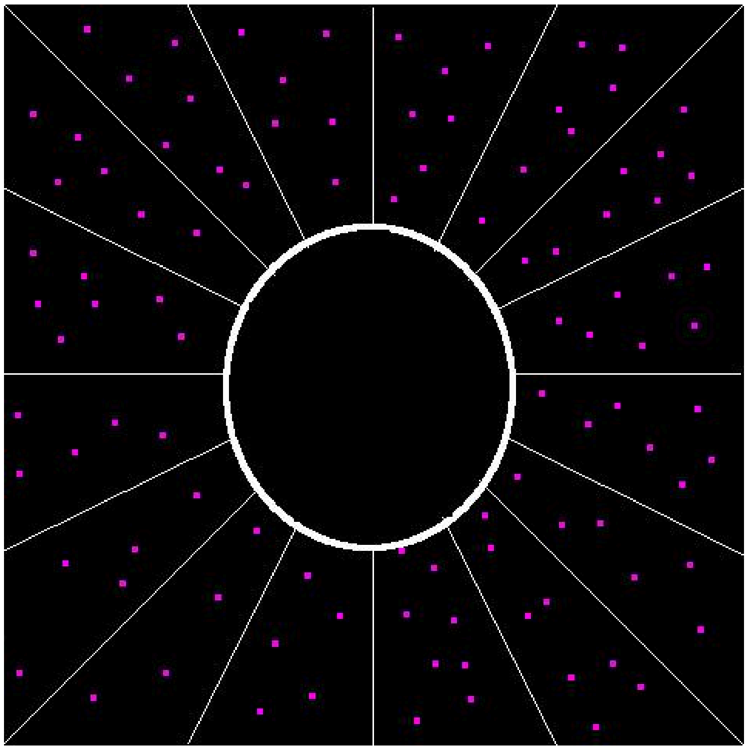


Figure 18. Background Image

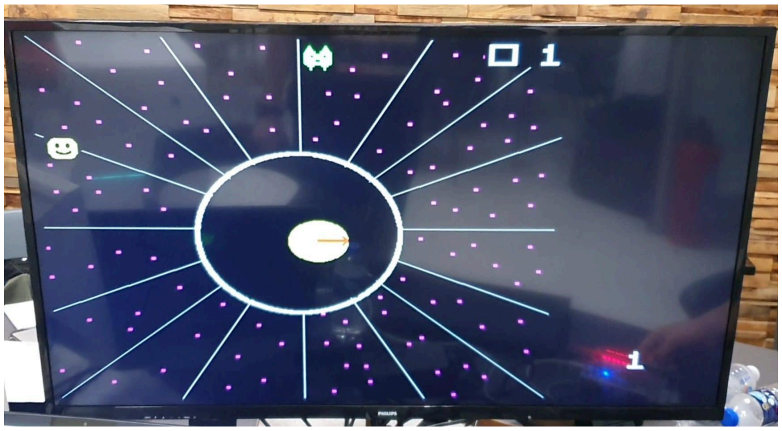
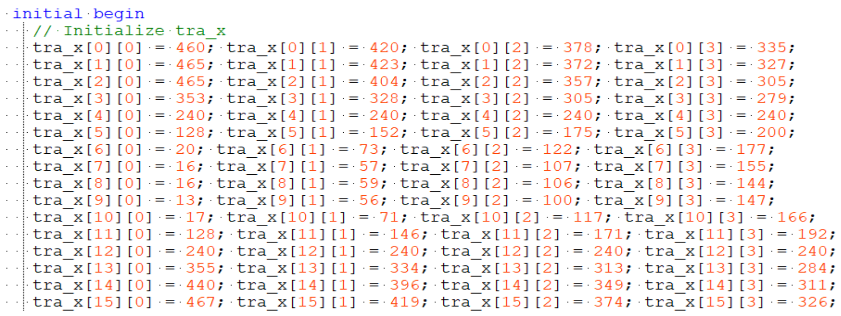


Figure 19. Game Screen



```
initial begin
    // Initialize tra_x
    tra_x[0][0] = 460; tra_x[0][1] = 420; tra_x[0][2] = 378; tra_x[0][3] = 335;
    tra_x[1][0] = 465; tra_x[1][1] = 423; tra_x[1][2] = 372; tra_x[1][3] = 327;
    tra_x[2][0] = 465; tra_x[2][1] = 404; tra_x[2][2] = 357; tra_x[2][3] = 305;
    tra_x[3][0] = 353; tra_x[3][1] = 328; tra_x[3][2] = 305; tra_x[3][3] = 279;
    tra_x[4][0] = 240; tra_x[4][1] = 240; tra_x[4][2] = 240; tra_x[4][3] = 240;
    tra_x[5][0] = 128; tra_x[5][1] = 152; tra_x[5][2] = 175; tra_x[5][3] = 200;
    tra_x[6][0] = 20; tra_x[6][1] = 73; tra_x[6][2] = 122; tra_x[6][3] = 177;
    tra_x[7][0] = 16; tra_x[7][1] = 57; tra_x[7][2] = 107; tra_x[7][3] = 155;
    tra_x[8][0] = 16; tra_x[8][1] = 59; tra_x[8][2] = 106; tra_x[8][3] = 144;
    tra_x[9][0] = 13; tra_x[9][1] = 56; tra_x[9][2] = 100; tra_x[9][3] = 147;
    tra_x[10][0] = 17; tra_x[10][1] = 71; tra_x[10][2] = 117; tra_x[10][3] = 166;
    tra_x[11][0] = 128; tra_x[11][1] = 146; tra_x[11][2] = 171; tra_x[11][3] = 192;
    tra_x[12][0] = 240; tra_x[12][1] = 240; tra_x[12][2] = 240; tra_x[12][3] = 240;
    tra_x[13][0] = 355; tra_x[13][1] = 334; tra_x[13][2] = 313; tra_x[13][3] = 284;
    tra_x[14][0] = 440; tra_x[14][1] = 396; tra_x[14][2] = 349; tra_x[14][3] = 311;
    tra_x[15][0] = 467; tra_x[15][1] = 419; tra_x[15][2] = 374; tra_x[15][3] = 326;
```

Figure 20. *Zoomed in version of Figure 5.*