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 42

ERHAN ALPAN - 2515500
OĞUZHAN OĞUZ - 2516631
HACER AYÇA YILMAZ - 2517308
AHMET TAHA ÇELİK - 2515831

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. To drive a VGA monitor, five signals are needed: HS(horizontal sync), VS(vertical sync), R(red), G(green), B(blue). They are consecutively 2 digital signals and 3 analog signals. The monitor displays the picture line by line, starting from the left top to the bottom right. To count the lines, ie. the pixels, "x" and "y" variables are used in the code as x generating HS and y generating VS. Since hsync is an active-low signal, applying a logical 0V pulse to HS starts the row display and ends it with the next low pulse. The picture is displayed between these two low pulses, ie. the display interval, through the RGB data that is specified in the code. VS works similarly as it also displays the picture between two low pulses other than vsync operates with the set of rows while hsync operates with pixels. The VGA display process is given in the Figure 1. In terms of the diversity of the shapes, 3 distinctive classifications were asked from us: spaceship, enemy and bullet. To meet the requirements, we displayed a boomerang shaped spaceship and three distinctive enemies with each of them having various health points. We displayed the bullets by coloring only one pixel and used different colors to indicate the different powers of the bullets. The damage dealt to enemies was also wanted from us by leaving the damage indication way undetermined. To meet this requirement, we showed the higher health point enemies step by step with fading pictures. (Figures 6-11) Some texts were also needed to be shown such as "Game over" and "Score". Although we first approached this situation as we did for the enemy pictures, it did not work and displayed mere colors instead of the text images. To solve this problem, we inserted the binary texts of the text images into the code. To get used to with the VGA implementation, we started from the basics by trying to display a fully white screen. Then we moved on with parsing the screen as we wanted and displayed a

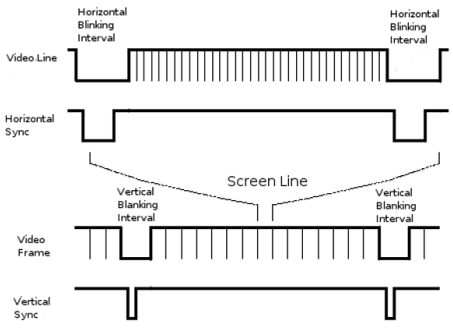

Figure 1. VGA display process

simple one colored spaceship in the middle. After these trials we started to build on our main display background and figures.

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.

Spaceship Control

Shape and Movement and Spaceship Controls on FPGA

A boomerang-shaped spaceship was chosen as seen in Figure 2. This type of spaceship was chosen for a more aesthetically pleasing appearance. However, during the coding phase, a 48x48 square hitbox was used for convenience and applicability. Additionally, images of the spaceship rotated 22.5, 45, and 67.5 degrees can be seen in figures 3, 4, and 5.

Figure 2. spaceship at 0 degrees

Figure 3. spaceship at 22.5 degrees

Figure 4. spaceship at 45 degrees

Figure 5. spaceship at 67.5 degrees

A variable with 4 bits namely Rotation is defined in the code. Each time the rotation keys are pressed, the value of the rotation changes by 1 depending on the input. Since it is defined with 4 bits, we have only 16 different values between 0 and 15. In the vga module, the spaceship is displayed depending on the rotation of it. The image of the spaceship transformed into a text file to store the color code of each pixel. With the help of $readmemh function of the Verilog, the data is imported in the code as a matrix. By using these matrices with appropriate indexing, the spaceship is correctly displayed. In order to rotate the spaceship, two buttons of FPGA are used for rotating counter clockwise and clockwise direction. On each positive edge of these buttons, the rotation is updated by one. With this implementation, if the user holds down the rotation buttons, the rotation will only change once.

Shooting Modes

In the shooting mode part, three shooting modes were created to fire each color (white, purple, blue). The main working principle of the shooting modes is that the damage decreases with increasing number of shots. The purple shooting mode only fires one bullet in one direction at a time with infinite power, i.e., it kills the monster independent of its current health. The white mode fires two bullets in five directions at a time. Each bullet corresponds to one health damage of monsters, i.e., The red and green monsters can be killed via two consecutive white bullets (red monster =1hp, green monster =2hp), but the yellow monster can not (yellow monster =3hp). In blue shooting mode, it fires two bullets in three directions each time. Each blue bullet corresponds to two health damage. Thus, two consecutive blue bullets (total 4hp) kill any monster. It's important to note that in purple shooting mode, the bullet has infinite power(momentum), i.e., it steals infinite hp from an enemy, and the purple bullet keeps going and hits new enemies. Apart from the purple shooting mode, in white and blue shooting mode, when a bullet hits an enemy, it steals 1hp from the enemy. Still, it disappears after hitting, i.e., that bullet does not affect new enemies (it can be thought of as the hitting bullet lost all its momentum with the impact of the collision with the enemy) this idea makes the game more realistic.

Controls of Shooting Modes on FPGA

One button is used for changing the shooting mode and another button is used for firing new bullets depending on the rotation and current shooting mode.

Switching Modes

In the code, a 2 bits variable is defined to store the current shooting mode. Each positive edge of the button, the shooting mode changes within a loop. Since there are only 3 different shooting mode, one state of the variable is not used. Shooting mode can have the values 00 -> 01 -> 10 -> 00, changing with respect to the posedge of the button.

The Main Logic of Deciding Damage Levels for Shooting Modes

The main logic is firing less bullets in less directions corresponds to more damage to make the game more realistic. For example, a purple bullet kills all the enemies it collides but it can be fired one direction at a time. Hence, it has more power and more risks.

Showing Projectiles

The each bullet is showed via painted pixel which is moving on the screen, its movement is made via painting a new pixel on the direction of movement with each clock edge ie painting the corresponding pixel with a frequency 30hz creates a moving bullet on that direction.

Enemy Dynamics

Spawning at Predefined Angles and Choosing the Predefined Angles

A direction between 0 and 15 is assigned to each enemy. Then, the initial positions are written by hand according to this direction. In order for the screen to be divided into sixteen equal directions, there must be a difference of 22.5 degrees between the two directions. Angles of 0, 22.5, 45, 67.5, 90 ... degrees were preferred, respectively. Since it would be difficult to determine the coordinates of 22.5 degrees, 25 degrees was used since tan25 = 2. The same applies to 67.5 degrees, 65 degrees was used for simplicity.

Achieving Randomness and Spawn Probabilities of Each Type of Enemy

64 bits length linear feedback shift register is used to achieve randomness of the direction and type of the enemies. In each clock pulse, it is shifted by one and new digit is determined by the xor tapping of arbitrary digits. In this way, pseudo-random number is obtained. With the help of proper if-else structure and LFSR, the enemies can spawn at random directions with random types. The spawning rate of the enemy with 1 health is twice as the others. To be precise, health 1 enemies can spawn with probability 50% while the others have the probability 25%. But overlap problem has not been fully resolved. In rare cases, enemies can overlap each other.

Number of Enemies and Spawning Time

The number of the enemies on the screen is tracked by a variable. If the number of the enemies is greater or equal to eight, no new enemy can spawn. If the number of the enemies is less or equal to two, a new enemy will spawn with random direction and type. To prevent the game from being too easy or too difficult, the enemy spawn frequency has been set to 1 per 2 seconds. When 300 points reached in the game, the difficulty increases, and the spawn frequency decreases to 1.5 seconds.

Types of Enemies

Figure 6. Enemy 0 with 1 hp

Figure 7. Enemy 1 with 2 hp

Figure 8. Enemy 1 with 1 hp

Figure 9. Enemy 2 with 3 hp

Figure 10. Enemy 2 with 2 hp

Figure 11. Enemy 2 with 1 hp

Three different types of enemies were created. The first enemy can be seen in Figure 6. It is red and has only one health point. The second enemy can be seen in Figure 7. It is green in color. It has two health points at the beginning. When a health point is lost, the enemy appears, as shown in Figure 8. The last enemy can be seen in Figure 9. The last enemy can be seen in Figure 8. It is yellow in color. It has three health points. When two health points remain, the enemy appears, as in Figure 10. When one health point remains, the enemy appears, as in Figure 11. Each enemy's location, direction, type, and health are stored separately. In case of a collision with a bullet, its health and type are updated. When the enemy dies, all of its data is reset.

Movement and Calculating the Trajectories

The direction and location of each enemy are stored separately as x and y coordinates. The stored data is updated with each clock pulse. For example, for an enemy coming from the right, the x coordinate value is reduced by 1 with each clock pulse. The spaceship is already in the center, so enemies moving towards the spaceship are moving towards the center. The x and y coordinates of each orbit are recorded and updated for each clock pulse. For example, for a 0 degrees trajectory, the y value decreases by one with each clock pulse. For a 180 degrees trajectory, the y value increased by one. For trajectories of 45 degrees and multiples, x and y values are decreased or increased by one every clock pulse. For 22.5 degrees, the x value is decreased by two and the y value is decreased by one with each clock pulse (or increased). For 67.5 degrees, the y value changes by two in each clock pulse, while the x value changes by one.

Player Score and Game Over Conditions

Scoring System

Each time an enemy is killed, points are earned depending on the enemy's health. An enemy with one life gives ten points, an enemy with two lives gives twenty points, and an enemy with three lives gives thirty points. A thirteen-bit variable was defined to store the score. The variable value is updated and saved in each clock pulse. A maximum of 9048 scores can be recorded. If this stage is reached, the score is reset to zero.

Checking Collisions Between Enemy and Projectiles and Calculating the Remaining Health of the Enemy

For each enemy, it is checked whether there is a bullet within a certain radius (15 pixels). If there are bullets, a collision is ensured. Enemies' coordinates and health were already recorded. In case of collision, the power of the bullet and the power of the enemy are compared. If the bullet power is greater, the enemy will be destroyed. If the enemy's health points are higher, the enemy's health is reduced as much as the bullet power and the enemy continues to move.

Game Over

For each enemy, the distance of x and y coordinates to (240,240) [spaceship coordinates] is calculated. If the distance is less than 24 pixels, a collision occurs. Since the player has one life, any collision will end the game.

Coding Approach

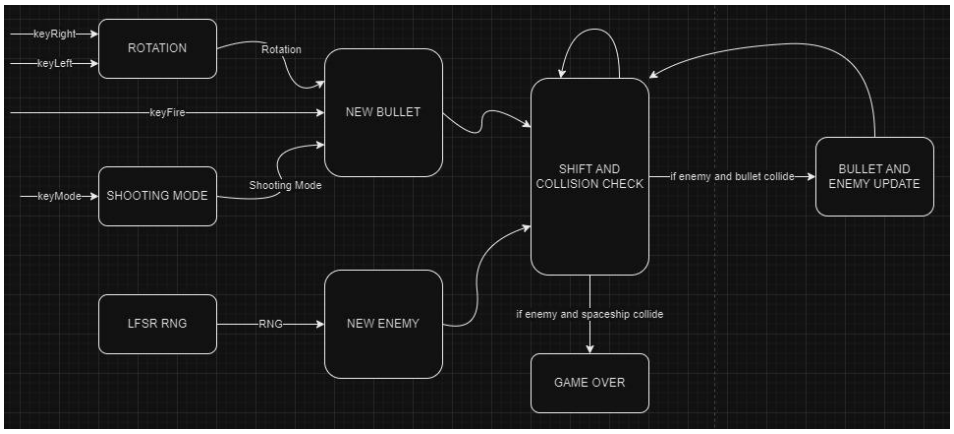

Figure 12. State diagrams of the code

To solve the problem, we divided the code into several parts. These parts can be seen in figure 12. We created enemies in certain directions using LFSR RNG. Afterwards, we determined the position of the spaceship and adjusted its rotations. We created bullets and shooting modes. New bullets are created according to rotational and shooting modes. Then, we instantly compared the positions of the bullets and enemies. We have defined the situations that will occur depending on the bullet power and enemy health point. We adjusted the ending of the game in case of a collision with the spaceship. So we actually divided the code into three main parts. Bullet part, enemy part and comparison system.

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. The first challenge was the display part of the spaceship. To solve this problem, we first determined a spaceship image. We then converted it to a text file containing the color code for each pixel in the image. We printed the space ship using this text file. Since we could not provide the 22.5 degrees, 45 degrees and 67.5 degrees situations from a single image, we created separate images and text files for each situation. The second challenge was making sure the enemies didn't overlap. Since enemy directions are random, in some cases, enemies may spawn in the same direction repeatedly, causing the images to overlap. We tried to reduce the frequency of enemy spawns to prevent enemies from spawning consecutively in the same direction, but this time, the game became much easier as the number of enemies in the game decreased. So, we could not fix it completely. In some cases, overlap may occur. The third challenge was to show the motion of the bullet. Each bullet represents a pixel painted on the screen. With each new clock pulse, a pixel in the direction of movement is painted white, and the old pixel is painted black. In this way, the movement of the bullet is shown.

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. Our game meets all the required criteria. Enemies with three different appearances and hit points are randomly spawned in 16 different dimensions. Enemies are moving towards the spaceship. Our spaceship in the center can rotate in 16 different directions and can fire in three different modes in each direction. The bullets are in three different modes and move in the desired direction. The movement of the bullets is visible to the eye. The collision of the bullet with the enemy can be detected. A collision is concluded by comparing bullet power and enemy health. For each enemy killed, 10 points are gained per health point. When 300 points are reached, the game level increases and becomes more difficult. If it collides with any enemy spaceship the game is over.

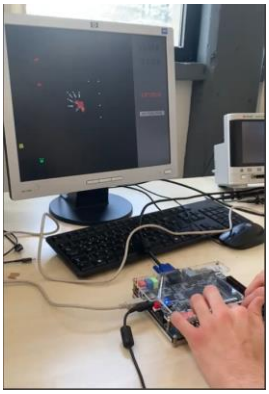
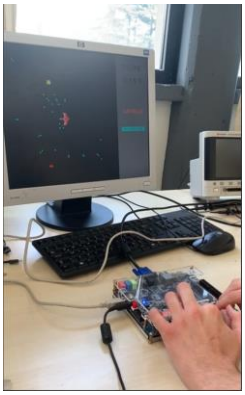

Figure 13. Two photos from the game

VII. CONCLUSION

Our results and solutions are consistent with design specifications. It's an enjoyable project to work on, as we can see that all of the theoretical knowledge coming from EE348 is used either directly in designing or creating ideas. Apart from that, we used the knowledge from the labs. We learned how to code efficiently in order to avoid memory problems in some parts of our design. Also, we learn lots of new things about Verilog and implementing our ideas in a creative and efficient manner.

A more complicated and enjoyable game design could be achieved if we had more time. We could have created more types of monsters and a more complex leveling system so that each level could include a boss fight. Each boss monster could have unique features. Apart from that, the quality of the gameplay can be improved. For example, we can add a motion to the spaceship. Also, with that feature, a more proper generation of monsters and their movement would be implemented. Therefore, adding motion to the spaceship would increase the difficulty and the fun of the game for sure. With more time and resources, the visual design of the monsters and the spaceship can be improved, and the animation of hitting a monster or killing it can be made more realistic. Also, by using more resources, even if it could have designed a two-player mode with the increased difficulty of the game, it would definitely cause more fun and motivation to play the game.

VIII. REFERENCES

[1] DE1-SoC User Manual, Terasic Tecnologies Inc. Jan 28, 2019. Accessed: June 1, 2024. [Online]. Available:

https://odtuclass2023s.metu.edu.tr/pluginfile.php/588923/mod_resource/content/0/2024%20Term%20Project%20Description.pdf

[2] "VGA". Accessed: June. 2, 2024. [Online]. Available:

https://www.fpga4fun.com/VGA.html

[3] "Javier Valcarce's Homepage". Accessed: June. 19, 2024. [Online]. Available: http://javiervalcarce.eu/html/vga-signal-format-timming-specs-en.html

IX. APPENDIX / APPENDICES

```
reg [1:0] modeF = 2'b00;
always @(posedge swF) begin
    if(modeF == 2'b10) begin
        modeF = 2'b00;
    end
    else begin
        modeF = modeF + 2'b1;
    end
end
```

Figure 14. Shooting mode changing code

```
// ROTATION ////////////////////////////////////////////
wire [3:0] R;
reg [3:0] R_R, R_L;
initial begin
    R_R = 4'h0;
    R_L = 4'h0;
end
always @(posedge keyR) begin
    R_R = R_R + 4'b1;
end
always @(posedge keyL) begin
    R_L = R_L + 4'b1;
end
assign R = R_R - R_L;
////////////////////////////////////////////////////////
```

Figure 15. Spaceship rotation code

```
for(integer i = 0; i < 8; i = i + 1) begin
   if(ENEMY[i].active) begin

      if(((ENEMY[i].x >= 10'd240) && (ENEMY[i].x - 10'd240 <= 10'd24)) || ((ENEMY[i].x < 10'd240) && (10'd240 - ENEMY[i].x < 10'd24))) begin
         if(((ENEMY[i].y >= 10'd240) && (ENEMY[i].y - 10'd240 <= 10'd24)) || ((ENEMY[i].y < 10'd240) && (10'd240 - ENEMY[i].y < 10'd24))) begin
            GAMEOVER = 1'b1;
            GAMEON = 1'b0;
         end
      end
```

Figure 16. Detection code of the spaceship and enemy

```
else begin
   for(integer j = 0; j < 40; j = j + 1) begin
      if(BULLET[j].active) begin
         if(ENEMY[i].d == BULLET[j].d) begin
            if(((ENEMY[i].x >= BULLET[j].x) && (ENEMY[i].x - BULLET[j].x <= 10'd15)) || ((ENEMY[i].x < BULLET[j].x) && (BULLET[j].x - ENEMY[i].x < 10'd15))) begin
               if(((ENEMY[i].y >= BULLET[j].y) && (ENEMY[i].y - BULLET[j].y <= 10'd15)) || ((ENEMY[i].y < BULLET[j].y) && (BULLET[j].y - ENEMY[i].y < 10'd15))) begi
                  if(BULLET[j].p == 2'b11) begin
                     ENEMY[i].active = 1'b0;
                     C_ENEMY = C_ENEMY - 1'b1;
                     SCORE = SCORE + (13'd10 * ENEMY[i].t);
                  end
                  else begin
                     BULLET[j].active = 1'b0;
                     C_BULLET = C_BULLET - 1'b1;
                     if(BULLET[j].p >= ENEMY[i].h) begin
                        ENEMY[i].active = 1'b0;
                        C_ENEMY = C_ENEMY - 1'b1;
                        SCORE = SCORE + (13'd10 * ENEMY[i].t);
                     end
                     else begin
                        ENEMY[i].h = ENEMY[i].h - BULLET[j].p;
                     end
                  end
               end
            end
         end
      end
   end
end
```

Figure 17. Detection code of the bullet and enemy

Vga Sync module:

```
module vga_sync (
    input wire clk,           // Pixel clock (25 MHz for 640x480 @ 60 Hz)
    output reg hsync,         // Horizontal sync signal
    output reg vsync,         // Vertical sync signal
    output wire video_on,     // Video on/off signal
    output wire [9:0] x,      // Current pixel x position
    output wire [9:0] y       // Current pixel y position
);

    // VGA 640x480 @ 60 Hz timing parameters
    parameter H_SYNC_PULSE = 96;
    parameter H_BACK_PORCH = 48;
    parameter H_ACTIVE_VIDEO = 640;
    parameter H_FRONT_PORCH = 16;
    parameter H_TOTAL = 800;

    parameter V_SYNC_PULSE = 2;
    parameter V_BACK_PORCH = 33;
    parameter V_ACTIVE_VIDEO = 480;
    parameter V_FRONT_PORCH = 10;
    parameter V_TOTAL = 525;

    reg [9:0] h_counter = 10'b0;
    reg [9:0] v_counter = 10'b0;

    assign x = h_counter;
```

```
    assign y = v_counter;
    assign video_on = (h_counter < H_ACTIVE_VIDEO) && (v_counter < V_ACTIVE_VIDEO);

    always @(posedge clk) begin
        if (h_counter < H_TOTAL - 10'b1)
            h_counter <= h_counter + 10'b1;
        else begin
            h_counter <= 10'b0;
            if (v_counter < V_TOTAL - 10'b1)
                v_counter <= v_counter + 10'b1;
            else
                v_counter <= 10'b0;
        end
    end

    always @(posedge clk) begin
        hsync <= (h_counter >= H_ACTIVE_VIDEO + H_FRONT_PORCH) && (h_counter < H_ACTIVE_VIDEO + H_FRONT_PORCH
+ H_SYNC_PULSE);
        vsync <= (v_counter >= V_ACTIVE_VIDEO + V_FRONT_PORCH) && (v_counter < V_ACTIVE_VIDEO + V_FRONT_PORCH
+ V_SYNC_PULSE);
    end

endmodule
```