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 48

Emirhan Yolcu - 2517332
Bora Özkan- 2516748
Alkım Bozkurt-2515674

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.

First of all, we need to cover the basic standardization of VGA. In this project, 640x480 resolution at 60 Hz rendering will be utilized. A video signal has some basics for both vertical and horizontal synchronization. Both horizontal and vertical screen area composes of a back porch, a front porch and a retraction signal other than the visible area.

Each line of video begins with an active video region, in which RGB values are output for each pixel in the line. The active region is followed by a blanking region, in which black pixels are transmitted. In the middle of the blanking interval, a horizontal sync pulse is transmitted. The blanking interval before the sync pulse is known as the "front porch", and the blanking interval after the sync pulse is known as the "back porch". (Ickes, n.d.)

Therefore, a video signal consists of 48 clocks of back porch, 16 clocks of front porch and 96 clocks of synchronization other than 640 clocks of visible are on the horizontal. On the vertical, there exists 33 cycles of back porch, 10 cycles of front porch and cycles of synchronization other than the 480 cycles of visible area. With a quick math, a clock of 25 MHz should be equipped in order to obtain the 60 Hz refresh rate on the screen

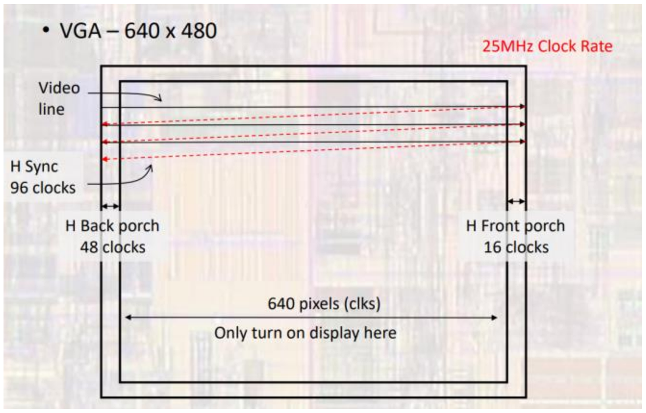


Figure 1 Horizontal video signal (Timoj, 2020)

.

$$800\,\frac{\text{clocks}}{\text{H}} * 525\,\frac{\text{H}}{\text{V}} * \left(\frac{1}{25^6\,\frac{\text{clocks}}{\text{sec}}}\right) = \frac{0.168\text{sec}}{\text{refresh}} \rightarrow 59.5\ Hz\ refresh\ rate$$

Equation 1 Refresh rate calculation (Timoj,2020)

Because of this standardization, a VGA controller code generally starts with the same declarations. In these basic implementations there are two counters (one for horizontal from 0 to 800, one for vertical from 0 to 525) that keep the information of on which cycle vertical and horizontal pulses are. For ease of use, we implemented these counters such that they also store the coordinate information on the visible area. It can be thought of as a cartesian coordinate system with left top corner as origin (0,0) and right bottom corner is the positive end (640,480).

After covering the VGA basics, all the objects are displayed on screen by assigning 8-bit RGB values. First, we started by painting the screen to some colors, then, we moved on painting specific pixel areas by using the coordinate system that is built in the previous part. With that being achieved, different patterns and objects could be drawn. Therefore, enemies and spaceship could be modeled. Different algorithms for different conditions are designed and different patterns are drawn on screen for these purposes. Below is a draft sketch on how we organized the screen.

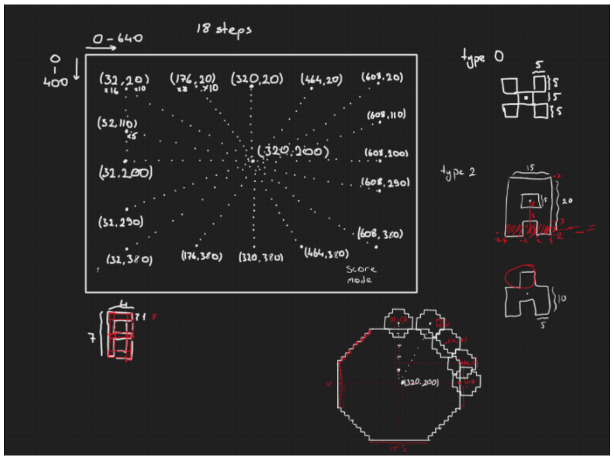


Figure 2: Screen organization and object modeling

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 backend module 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.

Shooter module is the code block where all logic operations happen. Enemy dynamics, shooting dynamics, spawn-death arrangements, key-button assignments all occur within this code block. The infrastructure and the building block steps of this module will be discussed.

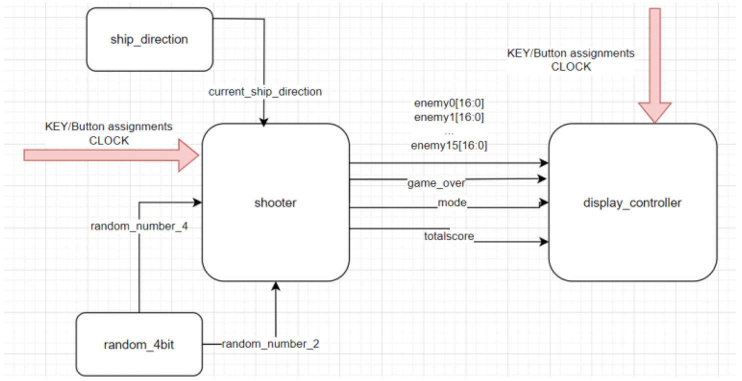


Figure 3 – Diagram for overall module structure

Spaceship Control

We have selected a symmetric base shape for the spaceship in order to minimize the complexity of displaying the spaceship based on the shooting direction. We only moved its front part in order to indicate the shooting direction, by using the selected direction information. The shooting direction is selected by user by pressing the KEYs on the FPGA board. A simple submodule called *ship_direction* is utilized.

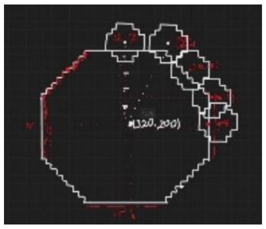


Figure 4 – Spaceship drawing,

One KEY increments the direction in CW direction and the other increments the direction in CCW direction. This module simply keeps track of two KEYs and yields the updated “ship_direction” information to main module synchronously. This information is kept within a 4-bit register as there are 16 directions at maximum. This information critical because it determines the display of the spaceship and the enemies which will be shot.

Shooting Modes:

We have 3 different shooting modes namely Mode 0,1 and 2. The selection of mode changes the damage dealt to the enemies. Therefore, we have implemented a case block and converted the information of "selected mode" into "damage dealt" and utilized that information. The mode information also determines the directions to be shot, therefore large case blocks are again utilized during shooting process.

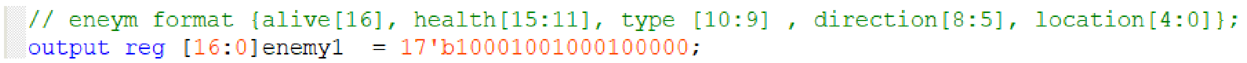
We had 3 modes, so a 2-bit register was sufficient to hold the mode information. Therefore, we have utilized two switches (SW [1:0]) on FPGA board to select the shooting modes. By changing switches, we were able to switch modes in real-time as the information coming from switches were implemented in a synchronous fashion.

Different modes were able to shoot at different number of trajectories. For example, Mode 0 was shooting at3 directions where Mode 2 shoots at 1 and Mode 1 shoots at 5. To balance the game, we assigned the lowest damage to widest mode and vice versa. The code block that converts the mode information into damage dealt information can be seen in Figure A1.

Lastly, we didn't have any moving projectiles. However, we knew the trajectories which are shot, and we knew if there exists a living enemy. By this two information we were able to indicate when an enemy is hit by a projectile. A grey square pops up in the center of the enemy display when the previously mentioned conditions are met. By this we demonstrated the damage dealt on enemies.

Enemy Dynamics

Before diving into enemy dynamics, the concept of "enemy" in our code should be explained. We have 3 types of enemies with different health. We have created 16-bit registers and defined them as "enemy". These registers hold the necessary information related to enemy such as health, location, direction, state of existence and type. All these bits are manipulated and utilized in the



```
// eneym format {alive[16], health[15:11], type [10:9] , direction[8:5], location[4:0]};
output reg [16:0]enemy1  = 17'b10001001000100000;
```

Figure 5 – 16-bit enemy register

following parts of the code. Since we had 16 directions, we have created 16 enemies. Figure X indicates the format of this register.

If the number of living enemies goes below 5, the enemy spawn process starts, and it never ends until the number of living enemies reach to the desired number 5. If we enter this spawning process, first we obtain two random numbers. If the number of living enemies goes below, a random number is picked. If there already exists an enemy, at that cycle no enemy is generated. If there exists no enemy in the direction of the random number, then we simply change the "alive" bit of the register at that trajectory and spawn an enemy there. Then we determine the type of the enemy with the help of Number 2, which automatically determines the health of the enemy too.

We have arranged 16 trajectories in the field. The display limitations were crucial when deciding the pixel ratios for trajectories. In the backend part, the static enemy registers were representing the enemies in the paths, so the trajectory information was kept within those registers

To provide randomness to our game we have used the previously mentioned random bit generator module called *random_4bit.* The trajectories and types of new-spawning enemies had to be random. For this purpose, a random number generator is used. A simple LFSR module with 2 taps was used to obtain this number. After generation, the two 4-bit and 2-bit random numbers are returned to the main code. We have tested the randomness of our module and verified that every single number appears at least once.

Enemy type determines the initial health of the enemies. Different type enemies spawn with different health. A table can be found below regarding the mentioned Appendix A2.

For each trajectory we defined a single enemy register. By doing this we have limited the number of enemies in a single path to 1. This automatically solved the problem of spawning multiple enemies on top of each other, nevertheless limited the gameplay.

For every internal clock cycle, the number of living enemies were checked with strict IF blocks. If number of living enemies is greater than 5 than no enemies were spawned and if it is less than 5 enemies were spawned. This logic kept the enemy number in control.

We have used a simple counter to control the enemy spawning rate. We have arranged the code as at every 1 second an enemy spawn if the number of enemies is sufficiently low. To be able to compatible with the display rate and to have a smooth gameplay we slowed down the clock for enemy spawning.

A table regarding the relation between enemy type, health and shape is provided in appendix.

The 16-bit enemy register has the information of "enemy type" within. By checking those 2 bits we have changed the shape to display and by this we have solved displaying different shape based on enemy type. For health, during spawn process the random number for type is incremented with 1 and multiplied with 2 to obtain the health related to type. Due to our levelling design, for every 10 points the level increases by 1. We also multiply the health of the enemies with level so as the level increases the health of enemies increase proportionally.

Again, by using a simple counter, we have matched our clock to refresh rate of display and within this block we moved the enemies. The 16-bit register also contains and information called "step" which simply defines the distance from end points of trajectories. For every new clock cycle, if the enemy is alive, we increase the step by 1. When location of any enemy reaches 17, it means it touches the spaceship and game over conditions get activated. The initial step is 0 and maximum step is 18. This number was calculated by using the regulations for display. We first determined the pixel number of the trajectory in display part. Then by also using the information of enemy shape size we have determined the pixel size for each step in a trajectory.

Then by these two we automatically calculated the maximum step size as 18. In other words, enemies spawn with "location = 0" and this number is increased by 1 every modified clock cycle until it reaches 17. The code block regarding enemy movement can be seen in Figure A3.

Player Score and Game Over Conditions

To develop a score system displayed in real time on the screen, we designed a system that will add 1 to the score every time an enemy is killed. Then, we converted this binary number into a BCD number and coded cases to display the ones and tens digits separately on the screen. To check collisions between enemy and projectiles mostly we use back-end code. The backend code operates based on the direction of fire and the firing mode, as previously mentioned. If there is an enemy in one of the directions being fired upon, a symbol indicating a decrease in health appears over the enemy, and the enemy's health decreases.

As seen in Figure 5, we store 5-bit health information for the enemies. Every time enemies take damage, their health decreases depending on the shooting mode. If their health drops to 0 or below, the 16th bit indicating that if they are alive or not becomes 0 and the enemies no longer appear on the screen.

During their lifetime, enemies move towards the central spaceship at a speed that varies depending on the level of the game. Each of them must move 17 steps to reach the center. As can be seen in Figure A3, if any of the enemies succeed in this, the game will be over. Figure A4 will appear on the screen and the last score obtained by the player will continue to appear on the screen.

Coding Approach

As can be seen in Figure A5, we created our code in a modular structure. All gameplay parts of the code are in the "Shooter" module. This module contains submodules that we use to determine the direction of the spaceship and to determine the direction and type of enemies completely randomly.

The "VGA controller" module allows information such as the appearance of living enemies on the screen, the current direction of the spaceship, the player's score, and which shooting mode is used, etc., depending on the information coming from other modules. Finally, we combined all the modules by determining the appropriate inputs and outputs in the "Shooter Project" module and obtained a working code structure.

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 to store enemy data. Information such as enemies' life status, current health, type, current step on the path etc. had to be stored. The first approach was to create a two-dimensional array. Every time an enemy is born, it was going to be appended to the array using a predefined module (or removed when died). Since multidimensional array implementation is illegal in Verilog in most cases, a different approach was developed. We created 16 different enemy registers (one for each path) that store the above-mentioned information. In the late-development, this also allowed us to directly implement further dynamics (such as enemy hit) since enemy name directly matches with the path number.

In the beginning, I was difficult to determine how everything was going to be placed and moved on the screen. In order to come up with a practical solution, the placements of objects were all parametrized. In order words, the drawing of a pattern was only coded once and then copy-pasted with different parameters. For example, see Figure A8 - Pattern for Enemy of Type 0. The drawing completely depends on *xpos2* and *ypos2* (for enemy on path 2) parameters. These parameters are defined with the step number on the path of the enemy (organization draft Figure 2 Screen organization and object modeling). Hence, they automatically move as the step number in related enemy's register changes. In another application, this method allowed us to write letters on screen by using our predefined seven segment-like pattern.

In the beginning, 4 bits were allocated for the health data on the enemy registers. This allowed enemies to have health of 16 at maximum. In the late-development, with the implementation of the new levelling system, enemies' health were able to exceed 16 at higher levels, causing them to have smaller health values. The problem is solved by also increasing the speed of the enemies with higher levels. This method made it impossible for a human player to reach such high levels.

VI. RESULTS

Although we had a sufficient output in the visual aspect of the game that met the criteria, if we had a more time, we could have used better looking pixel drawings for the spaceship and monsters. We would also have the opportunity to improve the animations in many ways.

There was only 1 major situation in the entire project that contradicted our goals on the first day. We designed that when fired in the game, bullets would come out of the spaceship and the bullets would collide with enemies and damage them on paths determined depending on the direction of the spaceship. Later, we realized that this situation would impose a huge workload on us. We also changed our plan precisely because we realized that the instructions given to us did not include the obligation to visually indicate the impact of bullets and enemy collisions. We planned to ensure that when a bullet hits the enemies, there will be signs on them indicating this situation, and we have successfully affected this in our code.Then, we made the necessary clock adjustments and started the simulation tests, as seen in Figure A6. When we realized that we got the results we wanted from these tests, we ran the game fully and obtained the final image in Figure A7.

VII. CONCLUSION

As a result, we created a code exceeding 7 thousand lines in total and successfully completed the project. During this process, we learned how to use Verilog in a wide range of environments and gained serious experience in hardware coding. To comment on the final output, although our code worked stably and successfully, we could not find rational enough solutions at many points because we did not yet have sufficient command of the Verilog language, which caused our code to become longer. If we wanted to make the code less hard code, we could use 2D matrices and include more for loops. We can also use pixel drawings to make enemy and spaceship appearances more beautiful. We could give players a better experience by allowing the game to be played using the keyboard.

VIII. REFERENCES

FPGADude. (2022). *Frogger_pt1*. GitHub. https://github.com/FPGADude/Digital-Design/tree/main/FPGA%20Projects/VGA%20Projects/Frogger_pt1

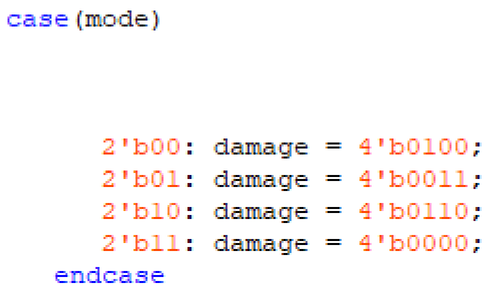
Ickes, N. (2004). *VGA Video (6.111 labkit)*. MIT - Massachusetts Institute of Technology. https://web.mit.edu/6.111/www/s2004/NEWKIT/vga.shtml

Timoj, J. (2020). *VGA basics*. Milwaukee School of Engineering. https://faculty-web.msoe.edu/johnsontimoj/EE3921/files3921/vga_basics.pdf

VGA Signal 640 x 480 @ 60 Hz Industry standard timing. (n.d.). Microcontroller VGA Interface projects. http://www.tinyvga.com/vga-timing/640x480@60Hz

IX. APPENDIX / APPENDICES

A1:



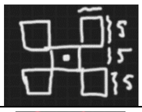
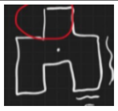
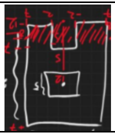
```
case(mode)


        2'b00: damage = 4'b0100;
        2'b01: damage = 4'b0011;
        2'b10: damage = 4'b0110;
        2'b11: damage = 4'b0000;
    endcase
```

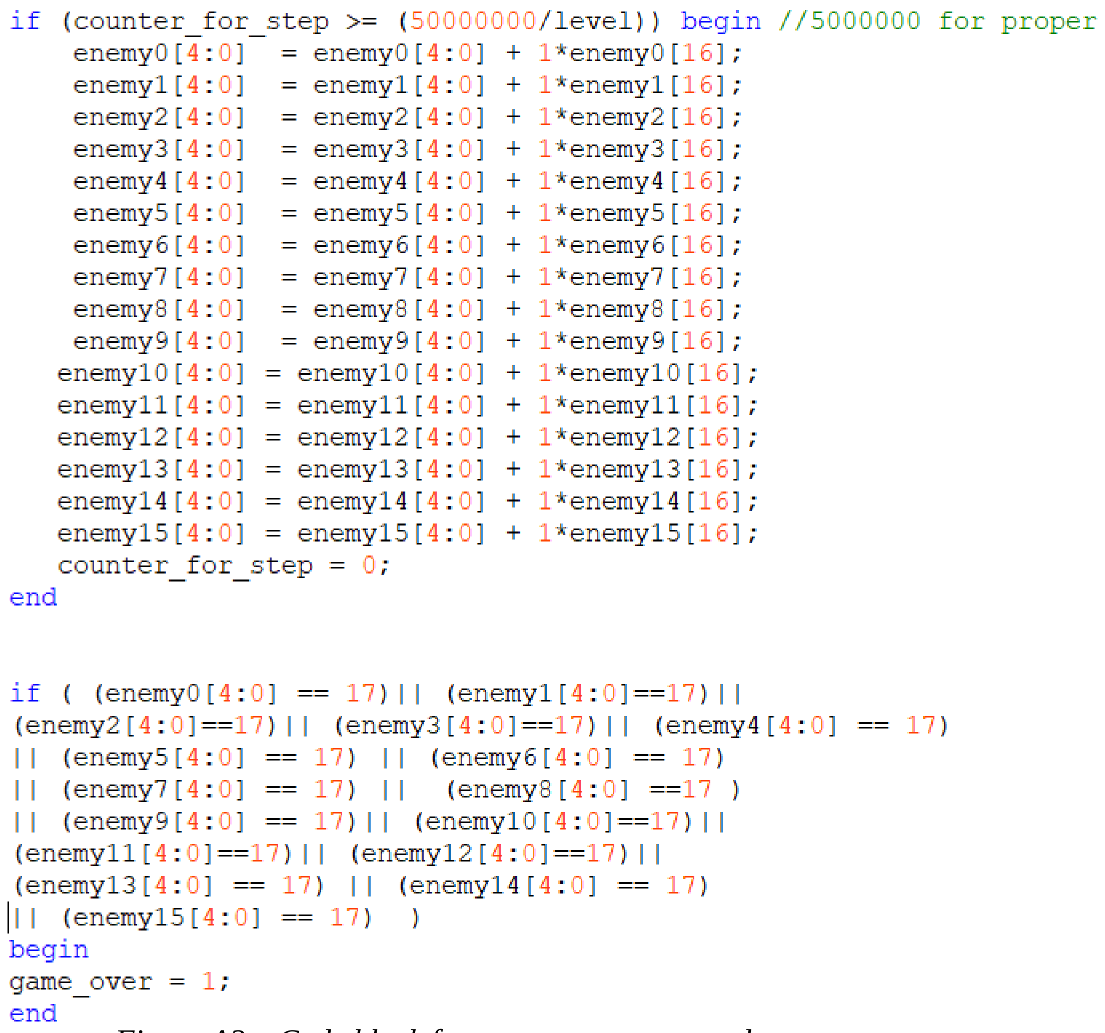
Figure A1 – Damage base on mode selection

A2:

Table A1: Table for properties of different enemy types

Type	**Health**	**Shape**
Type 0	2 Units	
Type 1	4 Units	
Type 2	6 Units	

A3:



```
if (counter_for_step >= (50000000/level)) begin //5000000 for proper
     enemy0[4:0]  = enemy0[4:0] + 1*enemy0[16];
     enemy1[4:0]  = enemy1[4:0] + 1*enemy1[16];
     enemy2[4:0]  = enemy2[4:0] + 1*enemy2[16];
     enemy3[4:0]  = enemy3[4:0] + 1*enemy3[16];
     enemy4[4:0]  = enemy4[4:0] + 1*enemy4[16];
     enemy5[4:0]  = enemy5[4:0] + 1*enemy5[16];
     enemy6[4:0]  = enemy6[4:0] + 1*enemy6[16];
     enemy7[4:0]  = enemy7[4:0] + 1*enemy7[16];
     enemy8[4:0]  = enemy8[4:0] + 1*enemy8[16];
     enemy9[4:0]  = enemy9[4:0] + 1*enemy9[16];
    enemy10[4:0] = enemy10[4:0] + 1*enemy10[16];
    enemy11[4:0] = enemy11[4:0] + 1*enemy11[16];
    enemy12[4:0] = enemy12[4:0] + 1*enemy12[16];
    enemy13[4:0] = enemy13[4:0] + 1*enemy13[16];
    enemy14[4:0] = enemy14[4:0] + 1*enemy14[16];
    enemy15[4:0] = enemy15[4:0] + 1*enemy15[16];
    counter_for_step = 0;
end


if ( (enemy0[4:0] == 17)|| (enemy1[4:0]==17)||
(enemy2[4:0]==17)|| (enemy3[4:0]==17)|| (enemy4[4:0] == 17)
|| (enemy5[4:0] == 17) || (enemy6[4:0] == 17)
|| (enemy7[4:0] == 17) ||  (enemy8[4:0] ==17 )
|| (enemy9[4:0] == 17)|| (enemy10[4:0]==17)||
(enemy11[4:0]==17)|| (enemy12[4:0]==17)||
(enemy13[4:0] == 17) || (enemy14[4:0] == 17)
|| (enemy15[4:0] == 17)  )
begin
game_over = 1;
end
```

Figure A3 – Code block for enemy movement and game-over

A4:

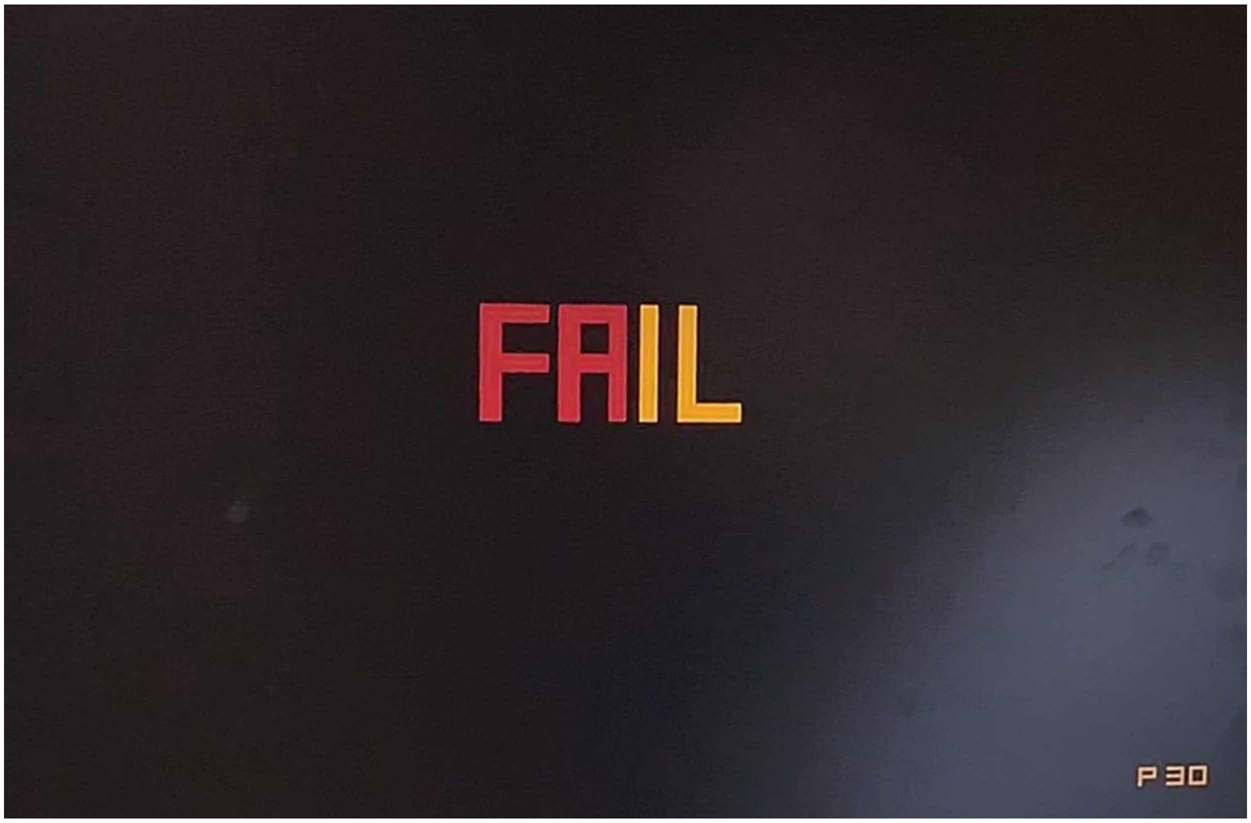


Figure A4 – Game over display

A5:

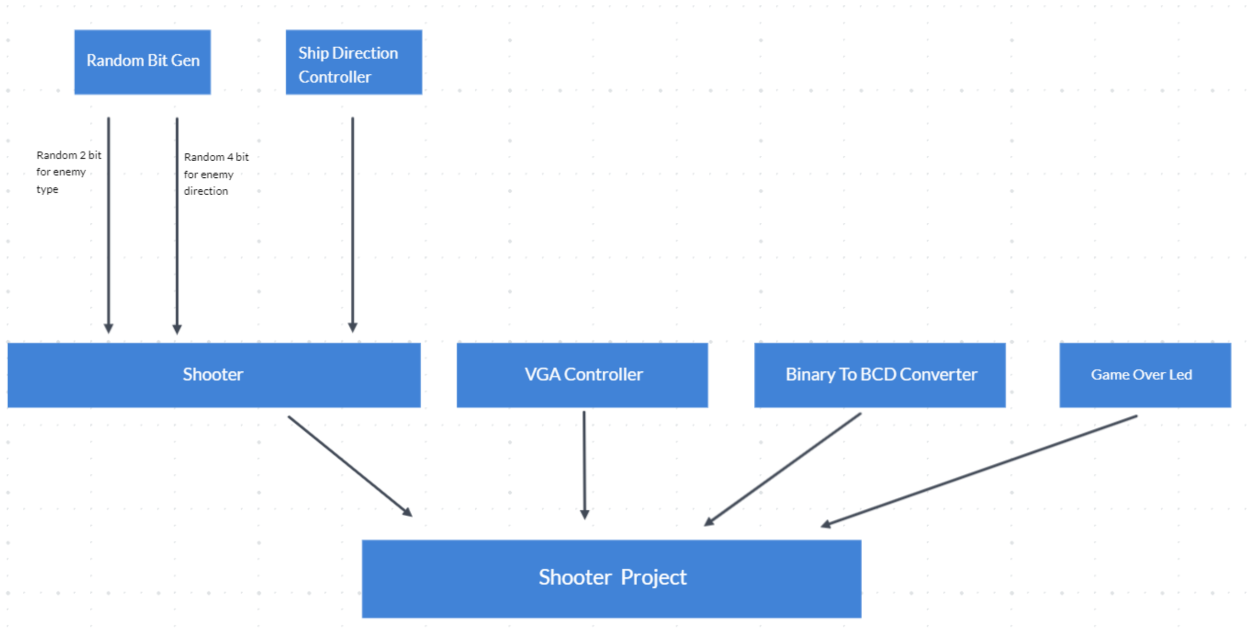


Figure A5 – Overall Code Structure

A6:

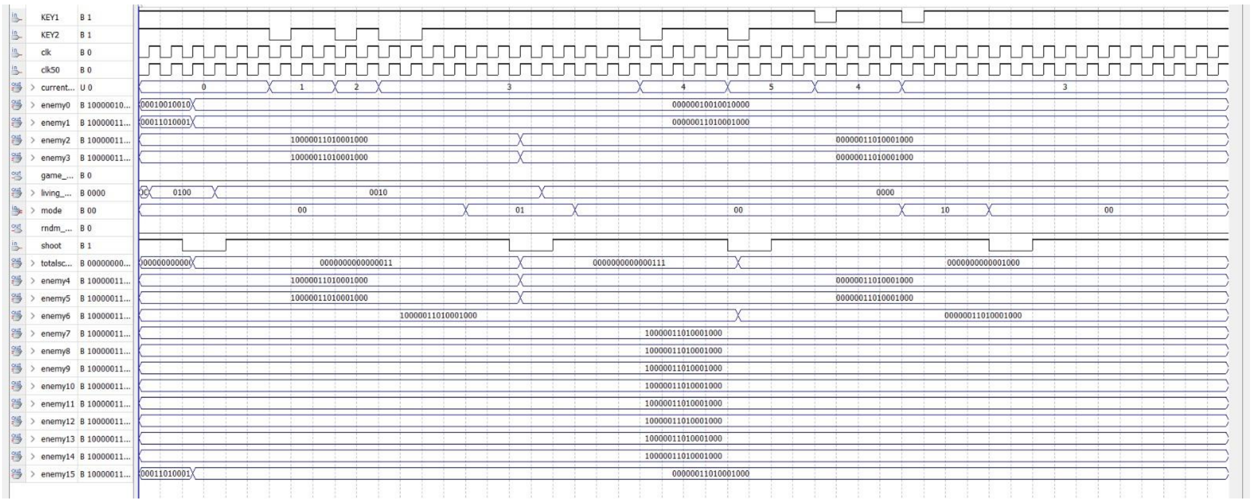


Figure A6 – Quartus Waveform Simulation Results

A7:

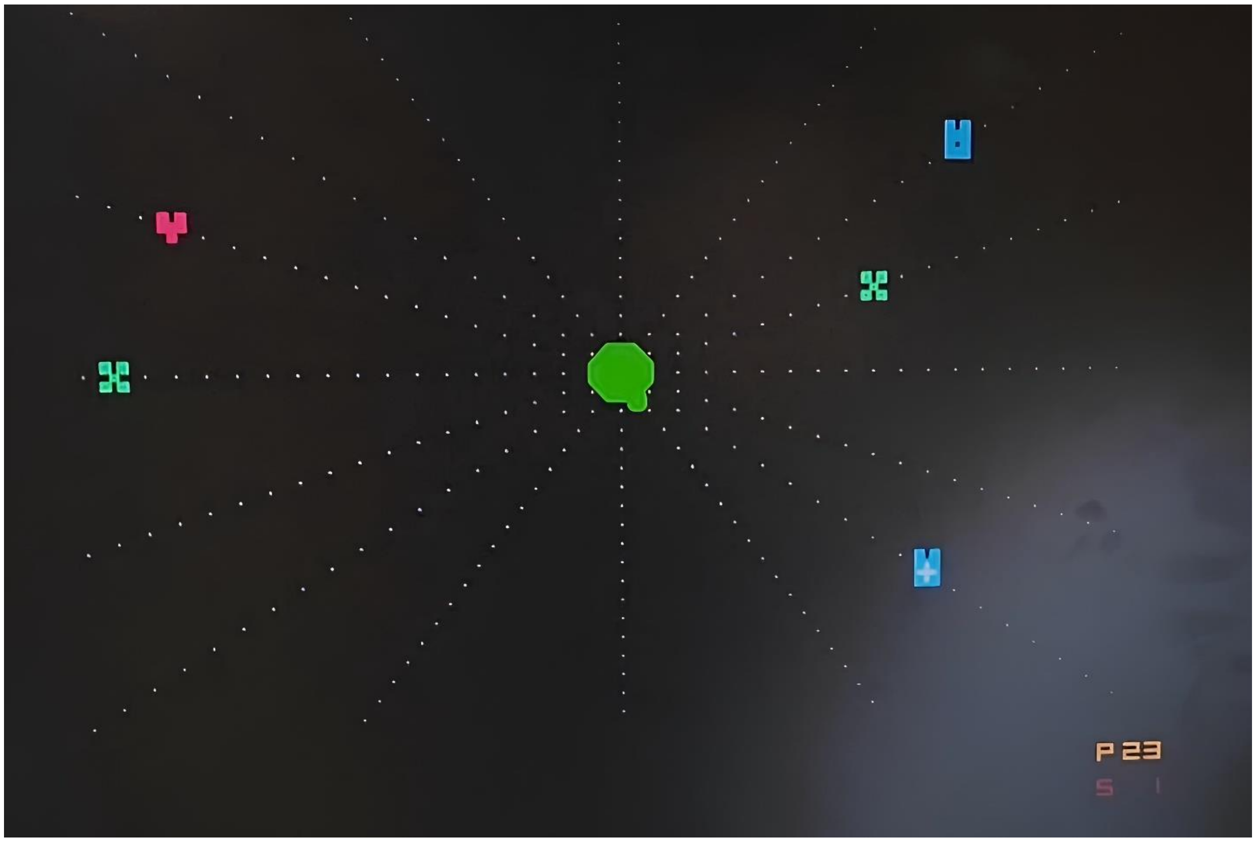


Figure A7 – Picture from working project

A8:

```
xpos2 <= (10'd608)-(10'd16)*enemy2[4:0]; ///origin declaration of the enemy
ypos2 <= (10'd20)+(10'd10)*enemy2[4:0];
if (y_count >= ypos2-7  && y_count <= ypos2-2)begin
       if((x_count >= xpos2-7 && x_count <= xpos2-2)||(x_count >= xpos2+2 && x_count <= xpos2+7))begin
              flag = 1;
              reg_red = 8'd21 + r_colorchange; ///type 0 color code // red span
              reg_blue = 8'd175;
              reg_green = 8'd253;
       end
end
else if (y_count >= ypos2-2 && y_count <= ypos2+2)begin
       if(x_count >= xpos2-2 && x_count <= xpos2+2)begin
              flag = 1;
              reg_red = 8'd21 + r_colorchange; ///type 0 color code
              reg_blue = 8'd175;
              reg_green = 8'd253;
       end
end

else if (y_count >= ypos2+2  && y_count <= ypos2+7)begin
       if((x_count >= xpos2-7 && x_count <= xpos2-2) || (x_count>=xpos2+2 &&  x_count<= xpos2+7))begin
              flag = 1;
              reg_red = 8'd21 + r_colorchange; ///type 0 color code
              reg_blue = 8'd175;
              reg_green = 8'd253;
       end

end
```

Figure A8 - Pattern for Enemy of Type 0