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 55

Cengizhan Başeğmez - 2529139
Mert Ekren - 2575173
Eren Erberk Erkul - 2591410

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 standard analog interface for displaying monitor graphics. It is widely used for various video applications, including gaming and general computer displays. Our VGA supports 640x480 pixels of resolution, although higher resolutions are possible with more advanced versions. Because VGA has timing constraints, we could not attain such resolutions with our clock. VGA timing involves precise synchronization signals, including Horizontal Sync (HSYNC) and Vertical Sync (VSYNC), which coordinate the drawing of pixels on the screen. In this part of the project, we aimed to display an image on a 2D screen, where the two dimensions are horizontal and vertical. Therefore, we required a way to keep the horizontal axis (each line) stable by defining the horizontal reference point and a way to keep each vertical axis (each column) stable. Both references were necessary to keep stability. We achieved that by defining Vsync and Hsync. These signals represent the next pixel to be printed and would be sent to the drawing module. Because of that, at every clock cycle, we would get an input of 24 bits, 8 bits of Red, Green, and Blue, each concatenated, representing the 24-bit palette, also known as natural color, which will represent the color of our current pixel. In this project, we were required to Establish a reliable method to render the game and visual elements on a VGA display, which required precise control over timing and synchronization signals to achieve a stable image. For signal stability, the accurate generation of HSYNC and VSYNC signals was paramount to ensure the display correctly interpreted the beginning and end of each horizontal line and vertical frame. Timing parameters, including the front porch, sync pulse, and back porch, were necessary to maintain the stability of our display.

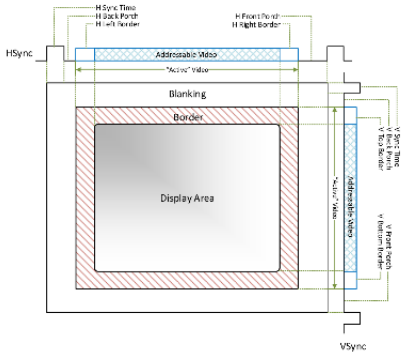

Figure 1 VGA Driver

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, how they work and interact with each other.

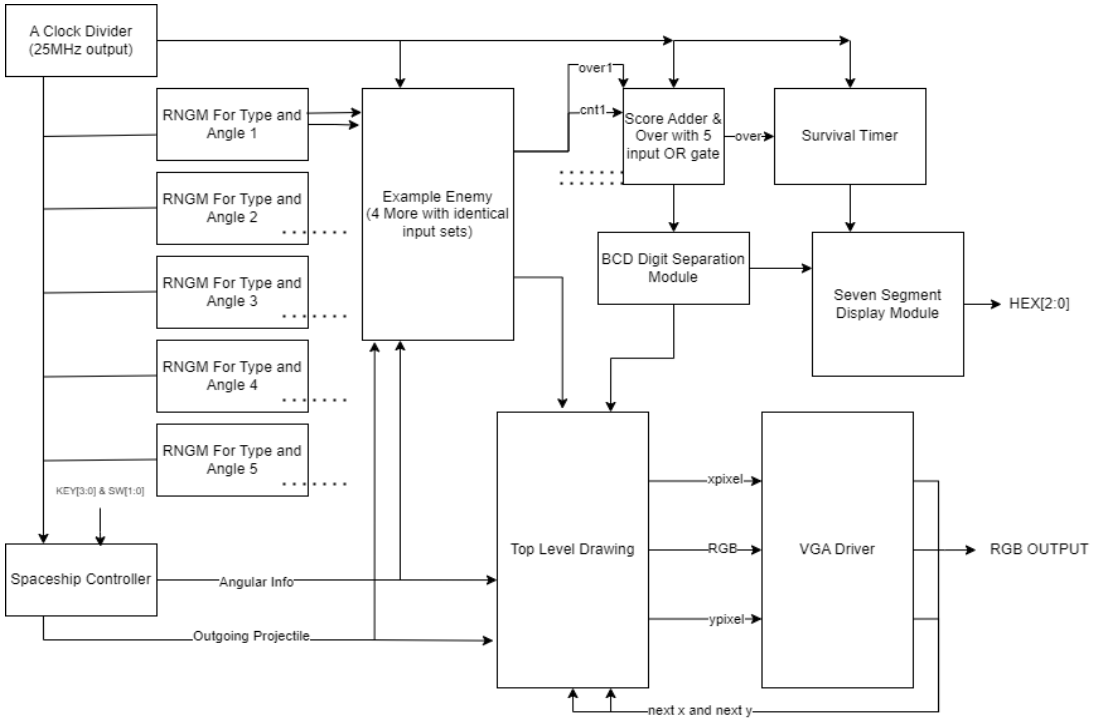

Figure 2. Overall Schematic

Our main display mechanism is under between top level drawing module and VGA driver, to understand better, here is the state diagram of our VGA driver shown below. Other than that, example enemy state diagram is also illustrated.

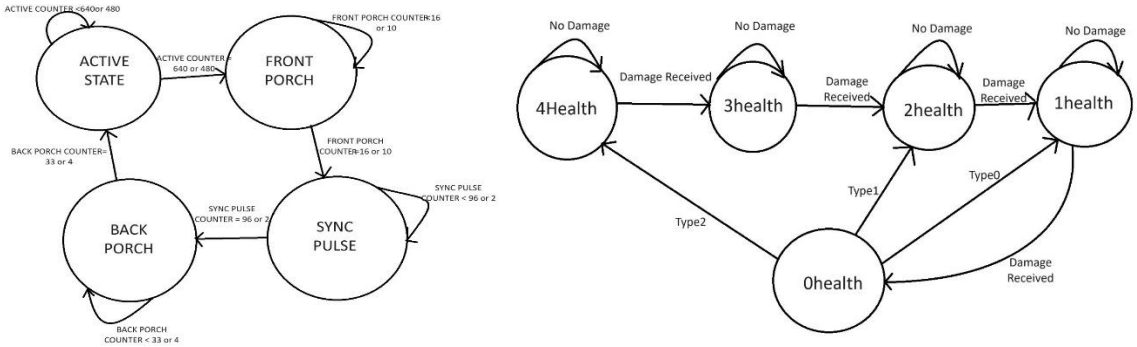

Figure 3 State Diagram of VGA Driver and Enemy

Spaceship Control and Shooting Modes

The shape we chose was that of a triangle tip with a base of the rectangle as can be seen in Figure 3. We decided on this shape to make the implementation more manageable for the FPGA. As was previously mentioned, we only used the embedded memory of the FPGA. For rotation, we used the 1st and 2nd buttons on the FPGA. The rotation mechanism is implemented using a synchronous counter. After it reaches 225ms, the rotation command is given, and depending on the input, it turns clockwise or counterclockwise. The counters reset to 0 if we reach 225ms. If we were not to press any of the buttons for 225ms, the counters would automatically reset to 0. With this counter, we solved the bouncing problem without a complex debouncer circuit.

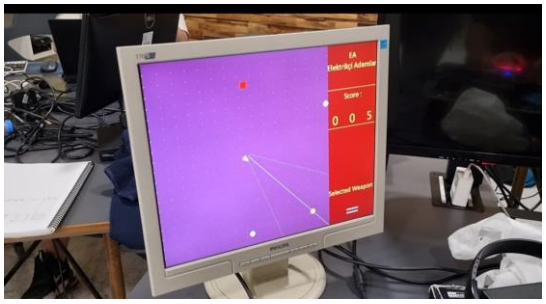

Figure 4 Spaceship Shootingmode2 pointing to 135°

The shooting mechanic was implemented using a button and two switches. The modes were implemented based on switches 1 and 0; if their value was 0, no damage was dealt; if it was '01' 1, the damage was handled over a 90° angle. If its value was '10 ', 2, damage was dealt for 45° angle, and if the switches were '11', dubbed by us as the execution mode named so because it dealt more damage than our strongest enemies' health and instantly killed any type. These damage levels were decided due to our enemies' health levels, which are 1, 2, or 4. Another aspect of our project was the animation of projectiles designed by mathematically calculating the angles and pixels on which the projectiles were shown for all 16 angles. Then, it is implemented for every shooting mode and angle input, as seen in Figure 3.

Enemy Dynamics

To achieve a symmetric display of the enemies, the screen is divided first such as the main game is on display ranges 0-480 vertical and 0-480 horizontal columns and rows respectively. The remainder part is for scoreboard which is located at 480-640 horizontal and 0-480 vertical not to leave no black pixels. After getting a square game field, 16 distinct, different angles should be chosen for enemy spawn points. Since 360/16 gives a 22.5-degree angle, then the rest is to calculate the points where they are initially located. The integer multiples of 45-degree angles were easier to find. To illustrate, if an enemy is coming from the up-right corner, its spawn pixel is (0,479). The table below illustrates

all the spawn-points pixel based with vertical, horizontal pairs. There might be 1 or 2 pixels miss clicks, which can be tolerated.

Pos.	0	22.5	45	67.5	90	112.5	135	157.5	180	202.5	225	247.5	270	292.5	315	337.5
H	240	340	480	480	480	480	480	340	240	140	0	0	0	0	0	140
V	0	0	0	140	240	340	480	480	480	480	480	340	240	140	0	0

Table 1. Spawn Pixels of Enemies

Creating true randomness is impossible; however, there is a technique which is covered in EE314, LSFR. These are the algorithms capable of creating pseudorandom numbers. Logic behind them is as follows; there exists an n-bit number which has an initial state. There are some locations randomly selected (user-dependent) which are called taps. Algorithm creates a feedback wire which is 1-bit XORed taps. This 1-bit information is given to the shift register, resulting in a new state. This cycle keeps the same way at every clock-pulse. An enemy module directly takes the inputs called “angle_random” and “type_random”, which is shown above with the Figure 1. Within the enemy module, these random inputs select the type and the spawn-point by the help of a case statement. Since enemy module is written in a generic manner, there is no difference in spawning. There is a critical point, which is obvious in one way. These "random" numbers are not actually random, they have a period. Tracking the random number generator for angle, it is guaranteed that there is no multiple spawning. That is, there is a generic enemy; however, there are multiple initiations for them. This method also allowed us to keep the enemy number under control. The logic is an enemy instantiation cannot be re-instantiated before it is dead. By using this fact, if there are 5 instantiations, there are constantly 5 enemies all the time. This method does not allow to call enemies within a time interval, which could lead the screen is overrun by enemies. That is why an enemy is spawned whenever the other dies.

The game implemented has three main enemy types. For three distinct types, game has three distinct shapes which also have three distinct health levels. Beginning with the health logic,health is represented by the color of enemy. If it has 4 health, it is white. If it is 3, then emits pink. For 2 and 1, colors are yellow and red respectively. In-game representations are shown below. At birth, square enemy has 1, cross has 2, bubble has 4 health bars.

Figure 5 An Example Collab of Enemy Types

When it comes to their code, it is not preferred to use HEX files since it requires some demanding spaces which slows down the compilation, it was chosen to apply bit-wise operation. Sprites are used for that, there is an example drawing for type-1 enemy, cross. That is, our enemy drawing module takes the inputs of type, health and center position of enemy. Type decides which sprite will be selected. Health is about the color as aforementioned. Center position will be discussed below

```
2'd1: begin
   sprite [0] = 16'b0000111111110000;
   sprite [1] = 16'b0000111111110000;
   sprite [2] = 16'b0000111111110000;
   sprite [3] = 16'b0000111111110000;
   sprite [4] = 16'b0000111111110000;
   sprite [5] = 16'b1111111111111111;
   sprite [6] = 16'b1111111111111111;
   sprite [7] = 16'b1111111111111111;
   sprite [8] = 16'b1111111111111111;
   sprite [9] = 16'b1111111111111111;
   sprite [10] =16'b1111111111111111;
   sprite [11] =16'b0000111111110000;
   sprite [12] =16'b0000111111110000;
   sprite [13] =16'b0000111111110000;
   sprite [14] =16'b0000111111110000;
   sprite [15] =16'b0000111111110000;
end
```

Figure 6 Example Drawing of Cross

Each spawn-point has a unique trajectory to the center (240,240). Naturally, the integer multiples of degree-45 are easier to find. To find the others, tangential approach is used. Since tan (22.5) gives approximately 0.4, an enemy coming from such an angle makes the movement 2 down 5 right, for example. Within the enemy module, with a divided clock of course, selected type and spawn-point enemy center position is updated at every clock cycle. To equate every angle's speed, following approach is used. Since 22.5 degree one covers approximately 5 or 6-pixel length at every cycle, 45 degree one should be updated in this way, like 4 down and 4 right.

Player Score and Game Over Conditions

Every enemy has its own counter embedded, that is, if the enemy output active goes low, counter increases by 1. Then, since 5 identical enemy modules are present, their individual counters should be summed. After that, a BCD Separation Module as shown in the Figure 1 gives the BCD digits. Then by bit-wise drawing of all BCD digits, a case logic draws the score.

Enemy and projectile collision is the one depending on a match between the shooting angle spray and the spawn-angle. If they match, and fire is on, there is a collector of damage. That is, the total damage is collected depends on the shooting type and angular match. We do not directly subtract this one from the health as there could be overflow problems. Instead, if this total damage dealt is less than the health, we prefer to subtract to avoid that problem. Simply, if it is higher than the local parameter type health, active output goes to low, unlocking the enemy module for new spawn. The enemy center point is shifted. Thus, if this center point is in a match between (240,240), the game over output goes high for a single enemy module. Then we OR all the enemy game over outputs.

Coding Approach

Beginning with the hierarchy, top-level is the one generated by System Builder. One below, there is our top-level drawing module where every piece is drawn. Almost every module feeds it. We preferred a modular approach since it is easy to debug. First layer is random number generators and they feed the 5 enemy modules. Enemy modules and spaceship controller works together and yield over condition, score. Score goes to the BCD Separation and is drawn. Over condition feeds the game-over drawing. Other than that, overall used inputs and outputs are CLOCK50, Switches, Keys and VGA RGB output, HEXs and LEDRs.

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.

At the very first times, we did not know that cosine and sine functions are not synthesizable. It did not take much time to realize it because it directly gave the error. Then we preferred to create a Lookup table; however, we again changed the approach for angle, which is aforementioned in previous sections.

Then we faced the main problem that took a bit longer. This was the random initiation problem. That is, when RNGM feeds the enemy module. There is an enemy created and it is moving. However, at the next clock cycle, this enemy changes type, erases itself from its previous position and selects another one. We had to lock our enemy module for an angle and type. This was done with a flag variable; we are forcing it to be low after the selection of spawn-point and type. In this way, the code never visits the selection part unless that enemy dies.

Third problem was about the score; when we die, if we kill an enemy, our score was increasing which had to be fixed. Then we added another else statement to the over module, like if the game is over, it does not add anymore.

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.

We successfully created an isometric shooter game on FPGA, which is fully operational, achieving the project's primary goal. In Figure SAFSDG, there are in-game and game-over screens for sample snapshots of the game. We displayed the game on a consistent 640x480 resolution using a VGA interface. The game's players are enabled with switches and the freedom to select different modes for varied projectile patterns and damage outputs using switches. Enemies with varying levels of health randomly appear at a predestined set of angles, which adds to the game's complexity. The scoring system functions smoothly as it records the number of enemy kills, where each kill is a single point. Thanks to the precise VGA synchronization signal generation (HSYNC and VSYNC), we maintained stable frame rates, ensuring smooth gameplay. Modular coding enabled us to find a reliable and efficient way to implement debugging and optimization. Also, the sound management of time constraints of the VGA interface provided an appealing display. We deployed a 24-bit color palate for a clear view of spaceships, enemies, and projectiles in graphics. In addition, during gameplay, the display remained stable, freed from glitches and flickering. Color-coded enemy health levels introduce strategic insights into the gameplay with critical functionalities such as spaceship rotation, varied shooting modes, and the mentioned score system. Future improvements would be a level system for the game, which could increase the difficulty as the player progresses. Also, unique power-up modes and more random enemy spawning could be implemented with a broader variety.

Figure 7 In-game Snapshots

VII. CONCLUSION

We created a fully functional isometric shooter game on FPGA. As discussed in the results section, the game provides a controlled gaming experience for the player with the implemented enemy dynamics and the scoring system. We translated our theoretical knowledge from our Logic Design Course and our basic implementations in the laboratory work to a more aimed project, improving our grasp of the material. Significantly, the challenges posed by VGA timing constraints and creating a more stable gameplay were both challenging and rewarding. With more time and resources, we would add a leveling system where difficulty increments as the player progresses in the game. We also would work on increasing graphics quality, implementing power-up systems, and increasing the variety of enemy properties for introducing more complexity to the game.

VIII. REFERENCES

1. H. Johnson and M. Graham, *High-Speed Digital Design: A Handbook of Black Magic*, Prentice Hall, 1993.
2. "FPGA VGA Graphics in VHDL," fpga4fun.com. Available: http://www.fpga4fun.com/FPGA_VGA.html.
3. P. P. Chu, *FPGA Prototyping by Verilog Examples: Xilinx Spartan-3 Version*, Wiley-Interscience, 2008.
4. L. E. Maissel and M. H. Glang, *Handbook of Thin Film Technology*, McGraw-Hill, 1970.
5. "VGA Controller," Verilog HDL Code from OpenCores, OpenCores.org. Available: https://opencores.org/projects/vga_controller.
6. J. P. Roth, "VHDL and FPGAs: VGA Interface," University of New Mexico, 2003. Available: https://ece-research.unm.edu/jimp/vhdl_fpgas/slides/VGA.pdf.