ECE 385

Fall 2019

Final Project Report

385 INVADERS

三百八十五 インベーダー

Pouya Akbarzadeh and Patrick Stach

Section: ABC

Lab TA: Vikram Anjur, Yuhong Li

Introduction

The goal for our final project was to re-design and implement the beloved game of Space Invaders using the FPGA as our System-on-Chip. While our game slightly differs from the original space invaders game which was released in the 1970s, it is still enjoyable, challenging, and obviously playable. The game's objective for us is pretty straight forward, eliminate the invaders before they eliminate you. In our version of the game, the invaders are some of our favorite course staff members. The game was achievable through a plethora of skills and knowledge gained through previous labs and lecture, meanwhile it also encouraged us to research some material that was briefly covered in class. This project also pushed us to think more creatively.

Written Description

Game Logic

In Space Invaders, you are controlling a ship in the X-direction with the left/right keys and shooting with the shoot key. Each level has a set amount of enemies that you need to destroy in order to proceed to the next level. Each of these enemies has a set difficulty which determines their number of lives, the shooting speed and the shooting frequency. Each level spawns an array of enemies that is more difficult than the previous - there are 5 main levels and one boss level at the end featuring 5 instances of Zuofu.

In order to win a level, you have to destroy all the enemies. These enemies continuously move left and right, then slowly creep down. If you let them creep down all the way to the ship, you automatically lose the game. Each bullet that you shoot that makes contact with an enemy will take off 1 life, and if they run out of lives they vanish. Each enemy can shoot provided they do not have any enemy blocking them. If one of their bullets hits you, you lose a life. If you run out of lives, you lose the game. When you destroy all the enemies in a level, no matter how far down they got to, it will respawn the next wave at the top starting position.

In summary: to win you need to pass all the levels by destroying all the enemies before you run out of lives. You lose if you let the enemies get all the way down to the ship or if you run out of lives.

There is a user interface that displays game stats: time since start of game (seconds), score, and number of lives. The score updated every time you hit an enemy: every hit adds their difficulty level to the score. For example, if every time you hit a “Varun”, the score will increase by 3 points. Since a “Varun” has 3 lives, completely destroying a “Varun” will give you 3x3=9 points.

If you beat the game, the screen will show the calculated final score, which is:

Score + Time Bonus + Lives Bonus =

Score + (500-Time) + 3 x Lives

Equation 1. Final Score Calculation. Time Bonus will be zero if Time > 500.

Below are tables of the statistics of enemies and each of the level configurations:

Enemy Name	**Difficulty Level**	**Shooting Speed (Pixels/Frame)**	**Shooting Frequency (% Chance /second)**	**Size (Pixels)**
KTT-Tech	1	2	11.76%	32x40
Jason	2	4	15.68%	32x40
Varun	3	6	19.61%	32x40
Vickram	4	8	23.52%	32x40
Mihir	5	10	27.45%	32x40
Zuofu	6	12	62.74%	64x80

Table 1. Enemy Statistics

Enemy Name	Round 1 Qty	Round 2 Qty	Round 3 Qty	Round 4 Qty	Round 5 Qty	Round 6 Qty
KTT-Tech	5	7	7	9	6	0
Jason	7	6	7	4	7	0
Varun	0	3	4	5	4	0
Vickram	0	0	3	5	5	0
Mihir	0	0	0	2	5	0
Zuofu	0	0	0	0	0	5

Table 2. Round Enemy Configurations

Project Features

Baseline Features

Before adding advanced features to improve the gameplay of our final project, we first implemented the baseline features to make our project a fully functional game. The first thing we added was in order to modify Lab 8 (ball bouncing) for our purposes was to constrain movement to only the X direction for the ship, and this is controlled by the keys on the FPGA: Key[3] for Left and Key[2] for right. We also fixed the boundary issues with the ball in Lab 8 where if the ball is all the way to the corner it will be stuttering back and forth. The new behavior matched that of the actual Space Invaders game: where if you move all the way to the edge of the screen the ship will just “stick” on the edge.

Next we implemented shooting: when you press Key[1] a bullet is spawned from the ship and moves upward until it hits the top of the screen, where it resets and you can then shoot

again. This behavior of only being able to shoot when the bullet is not on the screen introduced a bug not in the original game: when the enemies are really close you would be able to shoot unreasonably fast because the amount of time the bullet on the screen was miniscule. We fixed this by adding a counter that would enable shooting only after half a second has passed.

Then we added hit detection for the enemy: when an enemy gets hit by a bullet it will “die” and both the enemy and bullet will disappear from the screen. We also added a movement pattern that matched that of the actual Space Invaders Game: the enemy will move left and right and then slowly move down the screen.

We then implemented implemented an enemy being able to shoot: every set amount of time an enemy will shoot a bullet and if the bullet hits your ship you lose a life. For the absolute baseline we only made it possible for one bullet to be on the screen at a time and that only the third row could shoot (to avoid the case of a collision from an enemy shooting another enemy). There was a pattern where only enemy shot at a time but the specific enemy that shot changed over time. The behavior was not random at this time.

Then we added hit detection with enemy bullets and the ship: when an enemy bullet hits the spaceship the bullet will disappear and the ship will decrement lives by one. After that we added a finite state machine for the game control logic in order to transition between the “Game Start” screen, the actual “Game”, a “Win” screen and a “Lose” screen. You transition from the start menu to the actual game after you press any button, then if you destroy every enemy you get to the win screen, and if you run out of lives or let the enemies get to the bottom of the screen you get to the “Lose” screen.

Advanced Features Implemented

We implemented a couple of features in order to enhance the quality of life of the game. We first implemented a score that constantly updates the score every time is hit and a timer that counts the amount of seconds that have passed since the start of the screen. This original score in addition to the time taken to complete the game is used in the final score calculation. Both the score and the time passed is shown on the screen in addition to the current number of

lives and current level. We displayed text to the display by mapping the specific letter and pixel it was on to a font_data ROM file.

Displaying internal signals score/time/lives/round on the display in a readable format (decimal, not hexadecimal) posed a couple of challenges.. We had to figure out an algorithm in hardware to convert a binary sequence to distinct signals corresponding to each place of a decimal number. For example, if our input signal was 5'b11010, we would have to convert that to a usable format: 4'b6 in the one's place and 4'b2 in the ten's place to represent 26 (decimal).. Because the input was 20 bits, or a max number of $2^2 = 1048576$, this was not a matter of hard-coding a few simple cases. In addition to that, we had to decode that specific decimal number and output the font code (address) for that number to render based on the sprite data.

For a more polished look, we implemented complex sprites for each of the different types of enemies. Each of the enemies had their own set of color data- 4-bit color palette which mapped each specific pixel to a color based on the 16 possible different colors based on the specific 4 bit code corresponding to that location. We also had a color corresponding to a transparent channel, in our case this meant the pink background behind the TA's heads. We added an extra condition that if the specific color code corresponded to a transparent color channel the color mapper would not render that pixel but instead the background behind it.

We increased the complexity of the game by not only having enemies of different difficulty, but also having the game overall get progressively more difficult. There are 6 different enemy types, and each of them has their unique own number of lives, bullet speed, and shooting frequency. There are also 6 different levels with completely different initial configurations when spawning the enemies at the beginning of the level.

Another way we increased the complexity of the game was by having each enemy have their own instance of the bullet they can shoot rather than just having one bullet instance that all 27 enemies share. (I.e. before only one bullet could be active at a time and that bullet would simply just be shot at a random enemies coordinates). Theoretically if no enemies were blocking each other and all of them shot at the same time there would be 27 bullets rendered on the screen at once, each with their own unique velocity based on the difficulty of the enemy.

Not only did we allow every enemy to be able to shoot their own bullet, but we also made the shooting logic more advanced by adding blocking logic. Before we had it so that only the third row can shoot, which would never run into any issues since the third row is never blocked. However, we now are able to have every enemy be able to shoot if there are in the bottom most active row of their specific column. Each enemy has their own condition checks to check whether the enemies below them are active (alive), and if they are blocking them it prevents them from shooting.

We also implemented randomization of enemy shooting and generation of the stars for the background. Based on the way we randomized the behaviors of the enemy shooting and star generation, we were able to achieve both unique behavior at every runtime of the game and unpredictable patterns that are impossible to distinguish by eye regardless of the fact that our LFSR produces a sequence of 255 numbers.

We added various visual effects to make it easier for the user to distinguish key events and game info. For the ship, we added a 30 frame-length animation that flickers the ship on and off when it is hit by a bullet. This makes it very easy to see when you lost a life, as it can get very hectic with all the bullets and the user likely won't be staring at the number of lives in the top right corner. We also added health indicators for every single enemy in every single level. We found that this was very important to implement for a quality of life change because it makes it a lot easier to see what enemies are close to being dead. This makes it easier to plan out your strategy- for example, if you see that there is a very difficult enemy but they are almost dead, it would make sense to simply go for them and finish them off.

The way that health indicators work is that every single time an enemy is it, the color mapper would darken all the pixels by an amount corresponding to the amount of lives it had lost. This added difficulty because not only would the color mapper would have to render specific color pixels based on the enemy type and specific pixel DrawX, DrawY was on, but it also had to darken each color channel based on how many lives the specific instance of the enemy had.

Advanced Features Not Implemented

We decided against implementing A/DC conversion with a potentiometer for controlling difficulty because it would have taken too long to get it working. It was very difficult to find documentation on Analog to Digital conversion with the Cyclone IV and we felt our time would be better spent implementing other features instead of scavenging for documentation/ information. We also did not implement a difficulty select at the beginning: “Easy”, “Medium”, “Hard” because we felt we should stay true to the original game where the difficulty simply increases as the levels get harder, and in terms of design content we would still be demonstrating there being different levels of difficulty, just implemented differently than originally imagined.

We also did not implement sound: while it would have been nice to have sound it was not crucial to our game as say a synthesizer would need it. The events that we would have added sound to: shooting a bullet, enemy being hit by a bullet/ being destroyed, ship hit by a bullet - were already very clear to tell with visual effects on the screen. It would have been a nice bonus but it would have added a lot of complexity and a lot of extra time considering not only would we need to figure out how to get audio drivers to work on the FPGA in order to generate the sound on speakers, we would also need to figure out where to store the sounds as they take up a lot of space, and we would need to know how to store the sounds to represent the specific makeup of frequencies over time.

We were also originally thinking about adding some external outputs such as lights that would correspond to the game, but this didn’t really at much new in terms of difficulty content or game content so we decided against it.

We also didn’t end up implementing enemy animations, that probably would have been the next thing we would have implemented had we had more time to finish the project. Animations would have required at least one extra set of color pallete data for each enemy, ideally 4 sets: shooting, base 1, base 2, and hit. It would have made the game look more polished but would not have changed the gameplay at all since it is already very clear to see both when the enemy or spaceship was hit.

High-Level View of Component Purposes

We divided up different functions and instances into their own modules. We have an enemy array module which stores 27 enemy modules- each of these enemies have a randomization module for shooting behavior and a bullet module they control when they shoot. As for controlling enemy movement, we have an enemy path module that determines enemy movement based on the time passed. This module also calculates score based on number of hits.

We have a spaceship module that the user controls the movement and shooting using the onboard buttons. Both the spaceship and enemy modules can control an instance of a bullet FSM, and they have internal hit detection logic which will cause the bullet to disappear and decrement its lives by one in the event of a collision.

We also have a color mapper module that will map the specific pixel to the correct color based on the current object at that pixel and what that specific coordinate in that image is supposed to map to based on sprite sheets. This can handle both basic sprites (1's and 0's) and complex sprites (4 bit color palette).

For our background of shooting stars, we have a module that has 16 instances of stars that are randomly generated and this module outputs to the color mapper whether the current pixel should be a star or not.

We have a state machine that handles game logic. It initializes in the START state and handles the transitioning to the game, transitioning from levels, transitioning to WIN/LOSE, screen, etc. This module also sends control signals to initialize a module.

We have 3 different counters that keep track of game time- we have a frame counter that is used for seeding randomization, quarter second counter which is used for determining enemy path, and second counter which is used for determining our main game time.

We also have two modules that handle the VGA protocol: one module generates a clock signal for the VGA controller, and the other module handles the raster-order movement of the beam when drawing pixels.

Flow of Implementation

We can describe the whole circuit by describing the inputs/ outputs of two specific modules: one is the brain of the game and the other is the heart of the game

The brain of the game is the game control FSM. When the game starts, it starts in the START state. To transition between states and control the other modules, it has control signals:

- Input: Reset - Restarts FSM to the START state
- Input: Next Level - Advances the FSM to the next state. Will cause START state to transition to ROUND INIT state to initialize the first round, and will cause ROUND to transition to the ROUND INIT state to initialize the next level
- Input: Lives/ Too Far Down- Either will cause the game to go the ROUND state to the GAME OVER state
- Output: Round- If Round>6 the game will go from ROUND state to WIN state.
- Output: Init: passed to enemy array to reset (respawn) enemies
- Output: display_ID: passed to color mapper to determine whether to display the start screen, win screen, lose screen

The heart of the game is the Color Mapper module. It interfaces with every instance that needs to be displayed on the screen. This includes:

- Input: Is Ship/ Is Bullet/ Is Any Enemy Bullet/ Is An Enemy/ Is Any Star: Basically this family of “Is” logic variables determines whether an object exists at that specific pixel, i.e. if “Is Ship” = 1 then the ship exists at that pixel. The first sort of check that is done when deciding what type of object is present, so it checks all of the “Is” Family
    - These signals are crucial for hit detection- for a hit to exist for example the ship and enemy bullet, the expression would be Is Ship & Is Any Enemy Bullet.

- Input: Current Enemy Lives/ Current Enemy Types: We get these signals from the enemy array. In the enemy array module, we have logic to determine at this specific pixel that is being drawn- which enemy instance is present? Based on that that index we can access the array that stores all the enemy lives/ enemy types and get the specific lives/ type of enemy based on that index.
    - The enemy type is used for determining what sprite data to access
    - The enemy lives is used for the health indicator: the more lives the enemy has lost, the darker they will appear on the screen
- Input: DrawX, DrawY: This signal is generated from the VGA controller and it indicates which current pixel is being drawn. This is important because all object positions are compared to the pixel, both for choosing what color to be drawn and for determining hit detection ("Is" family relies on DrawX, DrawY)
- Input: Curr_Enemy_(X,Y)_ctr, Ship_(X,Y)_ctr: These are the calculated offsets in order to determine what pixel of an object we are drawing: (DrawX- X_Pos, DrawY - Y_Pos). These values are used to determine which part of the sprite data we are accessing
- Input: Score, Level, Game Time, Lives: Game logic from FSM and from enemy ship, these are displayed when the game runs
- Input: display ID: each type of screen has its own identifier, and this identifier is used to determine what elements should be drawn.
    - I.e.: If we are in the title screen we do not need to draw the ship
- Output: VGA_R, VGA_G, VGA_B: These are the only outputs of the color module but they are very important: These decide the intensity of each color channel for the current pixel being drawn.

Note that internally here we make all of our instances of text objects here with sprite_block modules, and these modules each output a "Draw" signal that determines if the current pixel being drawn is part of the text being drawn. That draw signal is used in the color mapper

Design Procedure

Overview

Originally we wanted to use the NIOS II processor for game elements that didn't need to be implemented instantaneously and for keyboard I/O.

However, we realized that including the NIOS II processor would add unnecessary complexity that we don't need in our game. For example: we only need a left arrow key, right arrow key, shoot button, and next level button to control our game. The FPGA already has 4 onboard buttons so there was no reason to use a keyboard thus we didn't need to use the USB I/O drivers.

Also, the elements that we wanted to incorporate using the NIOS II using a C program could be seamlessly integrated into our FSMs. For example, for score we have a signal that gets updated in the event of a collision with a ship bullet and enemy. For time, we have a hardware module that uses the frame_clk to compute how many time units passed (for our game time that we display to the screen we parameterize the module to count seconds). The ship lives are already implemented in the ship state machine- it has an internal signal that is initialized to 3 (lives) and is decremented in the event of a collision with an enemy bullet and ship. Our game state machine keeps track of what level it is on with an internal counter that increments after it detects a level has been finished.

We have a total of 4 different types of state machines: game logic (x1), ship (x1), enemies (x27) and bullets (x28). We decided that state machines were necessary because each of these instance had very obvious states, and those states have very specific behavior. For example, with the game logic we are either at the start screen, the game screen, the lose screen or the win screen. All of these screens have very distinct behaviors can it makes sense to use which state we are in both to determine what objects to draw and enable. Enemies and our ship also have obvious states: either they should be alive, they should be dead, they should be recovering from a hit, or they should be being initialized. These states have very important behaviors- ALIVE should enable the object, DEAD should disable the object (or end the game if it is the ship), INIT should reset the object (i.e. spawn enemies), and HIT should make the object

lose a life. Similarly, a bullet should either be active, initialized, or inactive. A bullet should be initialized on hardware reset, when a bullet is inactive it should not be involved in hit detection/ not be drawn but should be following the controller, and when a bullet is active it should be traveling vertically until a collision or it goes off the screen.

Even randomization was implemented completely in hardware using LFSRs that are seeded based on time intervals that vary from each run.

Simply put, we had no purpose for the NIOS II and adding it would add additional issues to consider. For example, the NIOS II has to be synthesized on the FPGA chip which obviously takes up a lot of space and would leave less room for our main hardware elements. Also, the code would likely have to be stored on SDRAM which present two caveats: you would need to use platform designer to generate the necessary modules to control the SDRAM and we would have to consider the timing of how long it takes for the memory contents to be valid.

We were originally thinking of storing our sprite map in SRAM or on-chip memory because we expected it to take up a lot of space. However, we noticed that taking no extra measure to use on-chip memory or SRAM and simply having the synthesis tool use registers and LUTs to store the sprite maps caused no issues at all. We had more than enough LUTs and registers left even with the rest of the hardware synthesized too. Compile time did not increase much either, and in terms of troubleshooting the actual game it was very easy to disable the sprite engine and use solid color squares for our enemies.

We attribute this to the fact that our sprites were relatively small. Fonts were 8x16 in dimensions (1’s and 0’s only for transparency). Even for color, we condensed all the colors into a 4-bit color palette (16 possible colors) for each TA, and each TA was only 32x40. Similarly, Zuofu had a 4-bit color palette and was only 64x80. Had we went more complex and had to store a 640x480 background image with more with RGB channel information instead of a color palette.

Sprite Rendering (Basic)

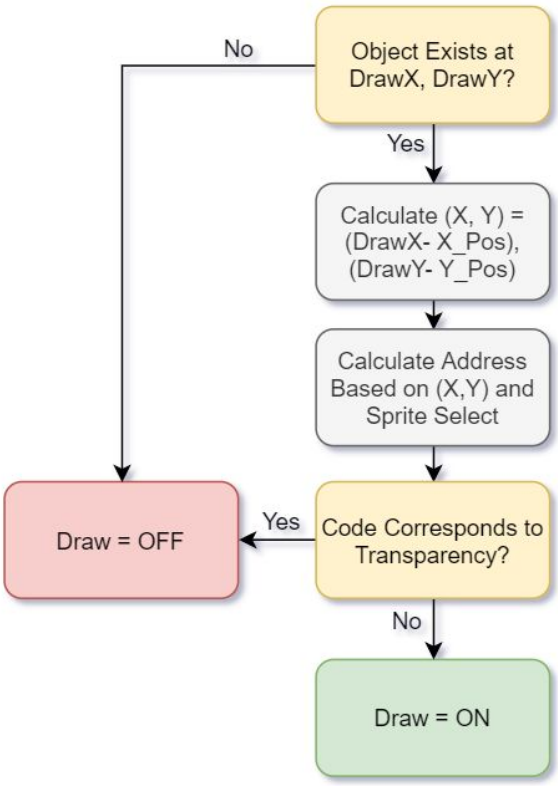


Figure 1. Procedure of accessing basic sprite data (just 1's and 0's)

The first type of sprite rendering engine was a basic sprite engine. The first task that the sprite rendering engine was determine whether a specific object exists at the current DrawX, DrawY coordinate. In the case of the basic sprite engine, we first check if an object that corresponds to basic text/ symbols exists in that location. If it does, we calculate what coordinates of the actual object is being rendered:

$$(X,Y) = (DrawX - X\_Pos, DrawY - Y\_Pos)$$

Equation 2. Equation to calculate coordinates of current pixel of object being draw..

. X_Pos and Y_Pos correspond to the top left coordinate of the object. In most cases the text we are rendering is multiple characters long, so the actual object width will be n x width, where n is the number of characters being rendered.

Not only do we calculate the coordinates of the actual object we are on, but we also need to calculate the coordinates of the specific character we are on and what ndex we are on, i.e. how many characters have already been rendered.

For example, let's say we want to render "385." Since each font character is 8x16 pixels, the object corresponding to "385' is (8 * 3) x 16 = 24 x 16. Let's say we at X coordinate 11. We calculate that we are at index 1 of the character we are rendering, so we are rendering "8", specifically X coordinate 3 of "8". We have an array of font codes corresponding to "385", so we access index 1 of that array to get the font code of "8".

The way the 8 x 16 font data is stored is by 8 rows of 16bit data. First to get to the "8" chunk of data, we have to use the font code to offset the starting location of our data to be the starting location of the "8" data. From there, we use the Y coordinate to offset the row, ie if the Y coordinate is 2 we would move down two rows from the start of the "8". Then, we use the X index to access the X'th bit of data: ie data[3], where data is the 2nd (index 0) row of data. We then return whether that pixel is a 1 or 0: if it is a 1 we render it, else if it is a 0 we don't.

Eventually as all (DrawX,DrawY) coordinates have been passed we have drawn the whole "385" text.

We can implement scaling of an image by an integer number, n. We do this by multiplying the object dimension we want to scale by that factor, then dividing the current coordinate of the scaling the original object in the direction by that number (integer division).

For example, if we want to scale the width by 2, we would make the object twice as wide, so “385” would not be 24 bits wide but 48 bits wide. Then to determine what coordinate of the font data we access, we have to do integer division on the X counter by 2. Essentially we will be drawing each pixel of the font data twice. I.e.: when we are at X coordinate 0, we will be grabbing the (0/2) = 0th x index of the font data, but if we are at X coordinate 2, we will be grabbing the (1/2) = 0th x index of the font data. We will essentially just be accessing the same piece of data twice consecutively.

Sprite Rendering (Complex)

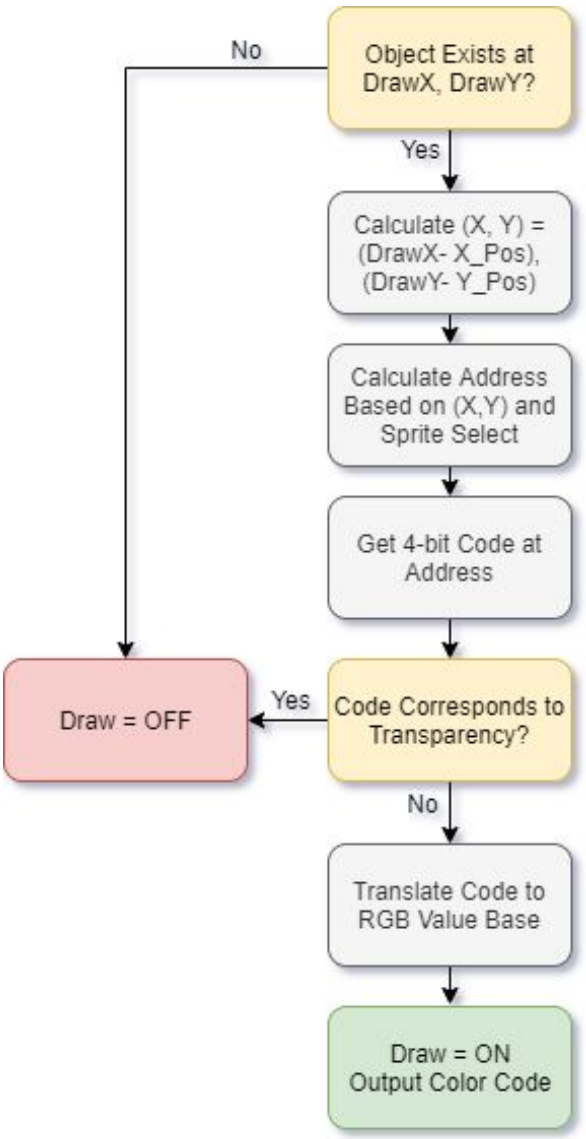


Figure 2. Procedure of accessing basic complex sprite data (4 bit color palette)

For complex sprites we are using a 4 bit color palette, meaning each pixel is mapped to 4 bits corresponding to a specific color. The procedure is nearly identical to basic sprite rendering except with a few extra conditions and exceptions. First of all, we are never rendering

consecutive enemies like we are consecutive letters, and although we technically could we do not scale the enemies - we map the pixels 1:1.

Although we calculate the coordinate of the object we are rendering like before, we are accessing a single 4 bit chunk of information that corresponds to that coordinate. We then need to decode that value: first we check if that value corresponds to transparency. If it does, we don't render the pixel. If it doesn't correspond to transparency, we translate the 4 bit color to an RGB color code and draw that color.

Randomization

We used randomization for both enemy shooting behavior and for generating shooting stars in the background. We implemented our randomization entirely in hardware using an 8 bit LFSR (Linear Feedback Shift Register). The way an LFSR works is that it is a shift register whose shift in bit is a linear function of its output bits. In our case, our function is:

$$Shift\ In = ((X8 \oplus X6) \oplus X5) \oplus X4$$

Equation 3. Equation to determine the shift in bit of our specific 8-bit LFSR.

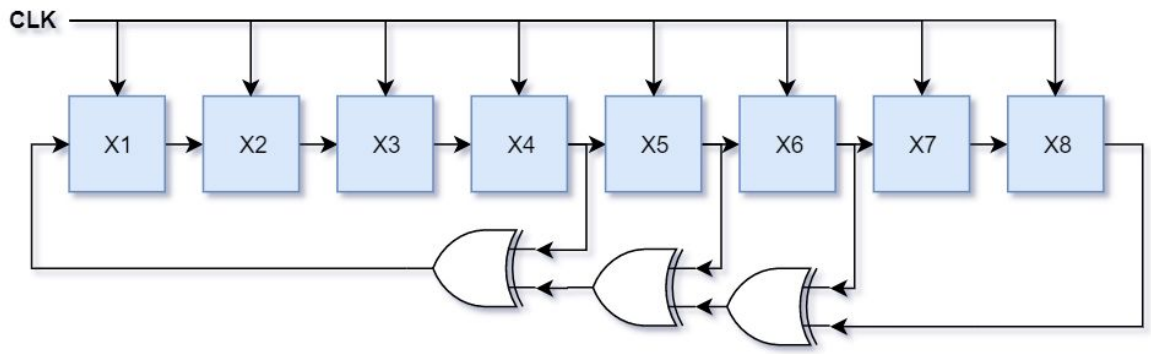


Figure 3. Circuit diagram of LFSR used in our design.

The initial start value is the “seed” which will determine the sequence of bits whose period is $2^n - 1$ states. In our case, $2^8 - 1 = 255$. In order to have random behavior we need to have a different seed for each instance of a run. We needed to use some sort of input that is different every single run, and that input we used was the number of frames from the system initialization to when the user presses a button to start the game (Round 1) or the time passed since the start of the game (Rounds 2-6). The specific enemy seed was a function in terms of the bottom eight bits of the number of frames passed and the enemy instance number:

Enemy Seed = (# Frames Passed [7:0]+10 x (Enemy Instance #))

Equation 4. Equation to determine seed of enemy for random behavior. A new seed is generated at the beginning of every round. The # Frames Passed for the first round is starting from system initialization, and for the other rounds it is starting from the actual start of the game.

We used a similar concept for randomizing the shooting star generation. However, if we only seeded at the beginning of every level it would be easy to see the pattern repeats. Not only is the value stored in the LFSR changing 4 times slower with enemies that with stars (every quarter second), the configuration of the enemies that are shooting constantly changes as enemies die out . Those two conditions make it impossible to determine by eye where the enemy shooting pattern repeats. This is not the case with the star generation.

In order to combat that for stars generation, not only did we load the seed at the start of each round but we also loaded the seed every 37 frames:

Load Seed = (Start round) OR (# Frames Passed % 37 ==0)

Equation 5. Equation to determine when to seed the star.

Similarly, the specific star seed was a linear function of the bottom 8 bits of number of frames passed and the specific star instance:

Star Seed = (# Frames Passed [7:0]+10 x (Star Instance #))

Equation 6. Equation to determine seed of star for random generation.

Our randomization module includes this configuration of an LFSR + seeding and outputs either 0 or 1 based on the probability value inputted. This probability value is from 0 to 255, and the equation to determine whether to output a 0 or 1 is:

Output =

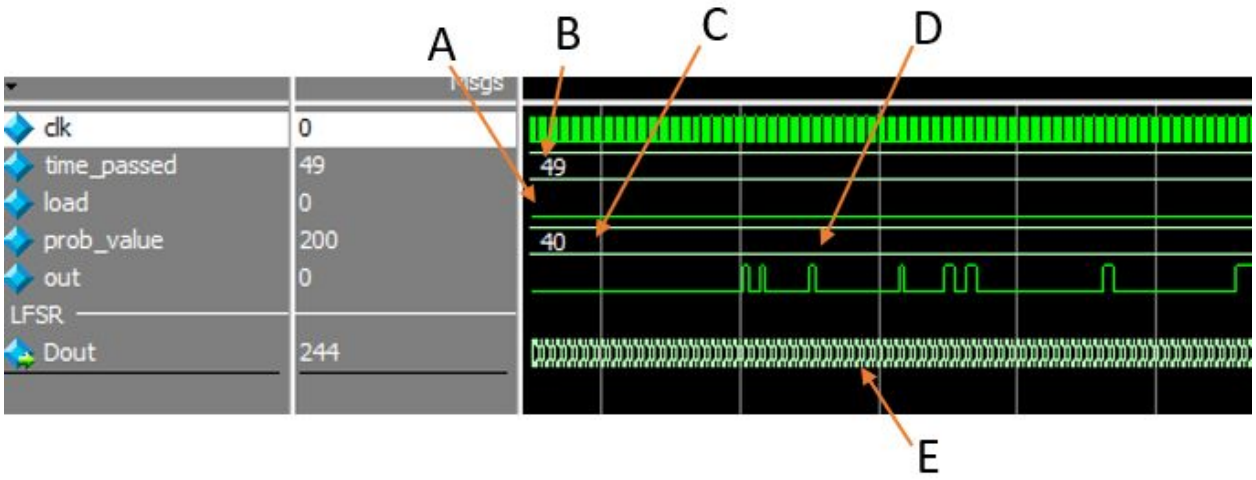
1: Value in LFSR < Probability Value

0: else

Equation 7. Equation to determine output of randomization module

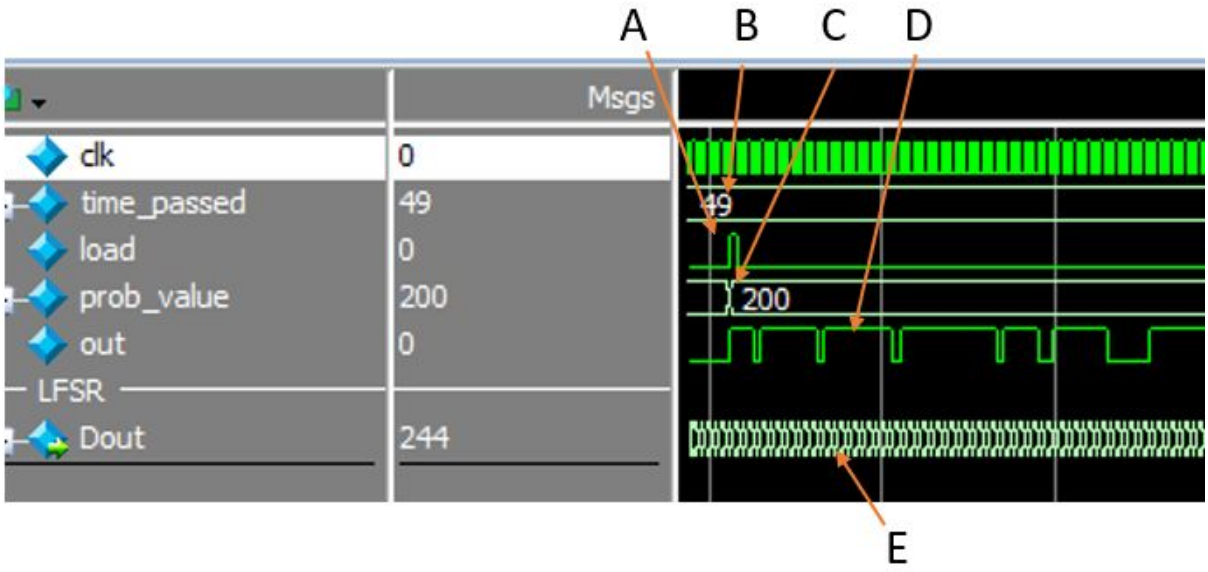
We can then see that the probability of outputting a 1 is (*Probability Value) / 255*. The value stored in the LFSR changes on the rising edge of the 50 Mhz clock, and the output of the randomization module is reloaded every second for enemies and every quarter second for stars. Stars and enemy bullets can only generated on the next clock cycle after that time period has passed.

Randomization Simulations



In this scenario, we see from A that the load signal is low so we are still retaining the original seed. From B, we see that in this case that the seed we used was the time that has passed. From C, we see that the probability value is 40 which on a scale to 255 is pretty low. From D, we see the output waveform occasionally goes high but not very often at all. In fact, the probability percentage is 40/255 = 15.68% which is a pretty accurate estimate of how often the

signal goes high. From E we see that the value in the shift register is constantly changing on the rising edge of the clock, and as that value in the shift register changes it will update the out value based on if the value in the register is less than the probability value



In this scenario, we see from A that the load signal goes high briefly so at that instance we are loading the shift register with the seed value. From B, we see that in this case that the seed we used was the time that has passed. From C, we see that the probability value is 200 which on a scale to 255 is pretty high From D, we see the output waveform is high most of the time but occasionally goes low. In fact, the probability percentage is 200/255 = 78.43% which is a pretty accurate estimate of how often the signal goes high. From E we see that the value in the shift register is constantly changing on the rising edge of the clock, and as that value in the shift register changes it will update the out value based on if the value in the register is less than the probability value

SV Module Descriptions

Shooting_stars

```
module shooting_stars(input logic Clk, Reset, frame_clk, frame_clk_re,
                      input logic [9:0] DrawX, DrawY,
                      input logic [7:0] fsec,
                      input logic load_seed,
                      input logic new_sec,
                      output logic [15:0] test_arr,
                      output logic is_any_star);
```

Description/Purpose: Shooting stars takes in graphics related signals in addition to inputs for randomization and generates a signal “is_any_star”, which determines if a star exists at that (DrawX, DrawY) coordinate. This module is used to generate the background of shooting stars by passing a control signal to the color maper to draw a star.

Enemy_array

```
module enemy_array(
  // For Base Enemy
  input logic   Reset, Clk, frame_clk, frame_clk_re,
  input logic is_bullet, // .*
  input logic [9:0] DrawX, DrawY, // .*
  input logic [9:0] X_Step, Y_Step, // .*
  input logic new_sec,
  input logic [3:0] level, // .*
  input logic load_seed,
  input logic [7:0] fsec,
  output logic [4:0] hex3_out,
  output logic is_an_enemy, // .*
  output logic is_any_enemy_bullet,
  output logic [8:0] LEDG,
  output logic [19:0] score,
  output logic all_dead,
  input logic is_ship,
  output logic [26:0] too_far_down,
  output logic [9:0] X_ctr,
  output logic [9:0] Y_ctr,
  output logic [3:0] curr_enemyType, // .*
  output logic [3:0] curr_enemy_lives); // .*
```

Description/Purpose: This enemy array contains 27 individual enemy instances and 27 individual randomization modules (for each enemy). It contains arrays for the configurations of the enemies based on level, and when it receives a Reset signal it will respawn the array of enemies based on the configuration chosen at the time of the preset. It also calculates the

score based on the number of hits and sends enemy information for the color mapper to process.

Rom_64x80

```
module rom_64x40 ( input logic [9:0] X_ctr,
                   input logic [9:0] Y_ctr,
                   output logic [23:0] RGB,
                   output logic draw_on
                   );
```

Description/Purpose: This module stores the sprite data for Zuofu, and the data is stored as a 4-bit color palette. It will output the respective color pixel based on what pixel of Zuofu we are rendering, and this is used by the color mapper.

Rom_32x40

```
module rom_32x40 ( input logic [9:0] X_ctr,
                   input logic [9:0] Y_ctr,
                   input logic [3:0] sel_32x40,
                   output logic [23:0] RGB,
                   output logic draw_on
                   );
```

Description/Purpose: This module stores the sprite data for the TAs, and the data is stored as a 4-bit color palette. It will output the respective color pixel based on what pixel of our TA and what type of TA we are rendering, and this is used by the color mapper.

Color_mapper_game

```
module  color_mapper_game ( input              is_ship, is_bullet, is_any_enemy_bullet,
                        input logic is_an_enemy,
                        input logic frame_clk_re,
                        input logic VGA_CLK,
                        input logic [3:0] curr_enemy_lives,
                        input logic [3:0] curr_enemyType,
                        input logic [9:0] curr_enemy_X_ctr,
                        input logic [9:0] curr_enemy_Y_ctr,
                        input logic [19:0] score,
                        input logic [9:0] Ship_X_ctr, Ship_Y_ctr,
                       input        [9:0] DrawX, DrawY,
                       output logic [7:0] VGA_R, VGA_G, VGA_B,
                        input logic [3:0] lives,
                       input logic [9:0] game_time,
                       input logic [3:0] level,
                       input logic [4:0] display_ID,
                       input logic next_level,
                       input logic draw,
                       input logic is_any_star,
                       input logic [6:0] total_num_enemies
                     );
```

Description/Purpose: This module handles all the color mapping of the game. It determines what color the specific pixel should be based on what type of object is at the specific DrawX, DrawY value, and what color it should be by either a specific chosen color for that object or chooses a color from the sprite sheet corresponding to that object. This module essentially “draws” everything and is responsible for what we see on the screen.

Game_fsm

```
module game_fsm
   (input logic next_level, reset,
    input logic [3:0] lives,
    input logic [26:0] too_far_down,
    input logic Clk,
    output logic init,
    output logic [3:0] round,
    output logic [4:0] display_ID);
```

Description/Purpose: This module is the heart of the game and controls the flow and operation of it. It interfaces with input signals and internal signals to determine the current state of the game and transitions, in addition to outputting control signals to control the modules in the top level such as enemies.

Bin_to_dec

```
module bin_to_dec
(input logic [19:0] in, output logic [3:0] out [6],
   output logic [6:0] out_code[6]);
```

Description/Purpose: This module converts a 20 bit signal into 6 discrete decimal places and outputs the font code for each place. This module is used for rendering variable values on the screen such as score, time, lives, etc.

Sprite_block

```
module sprite_block
#(parameter num=1)
(  input [9:0]   DrawX, DrawY,
   input  logic [9:0]  X_Pos, Y_Pos,
   input [9:0] Width,
   input [9:0] Height,
   input [6:0] font_sel[num],
   input [4:0] x_scale, y_scale,
   output logic pixel_on
);
```

Description/Purpose: This module interfaces with the basic font sprite engine to draw consecutive characters from an array of font codes while only needing to instantiate a single object

basic_font_sprite _engine

```
module basic_font_sprite_engine
# (parameter num=1)
   (input logic [9:0] X_ctr,
   input logic [9:0] Y_ctr,
   input logic is_obj,
   input logic [6:0] font_sel[num],
   output logic pixel_on);
```

Description/Purpose: This module interfaces with the Font_rom module to automatically automatically calculate the address and index of the data to access in order whether to draw a pixel or not based on which pixel of object we are rendering and what font code we are accessing)

Font_rom

```
module font_rom ( input [10:0]   addr,
                  output [7:0]   data
                );
```

Description/Purpose: This is basic font data stored as ROM, used by the color mapper to generate characters A-Z and 0-9 for UI such as titles, score, lives, etc.

Pos_edge_det

```
module pos_edge_det (input logic in, Clk,
                     output logic out);
```

Description/Purpose: This module detects the positive edge of signals. This is useful in cases where you only want to set an action to happen on the period immediately after the rising edge of the input signal, ie finding the rising edge of the frame clock.

Shift_register

```
module shift_register
   #( parameter N =8)
   (input logic [N-1:0] Din,
      input clk, reset, load, shift_en, shift_in,
      output logic shift_out,
      output logic [N-1:0] Dout );
```

Description/Purpose: This is a shift register that is by default parameterized to 8 bits. We instantiate this module to make a linear feedback shift register, used for randomization.

Random_generator

```
module random_generator (input logic [7:0] seed,
                         input logic clk, load,
                         input logic [7:0] prob_value,
                        output logic out);
```

Description/Purpose: This random generator uses an LFSR to output a value 1, (prob_value/255) x 100% of the time and a 0 the rest. We use this to randomize enemy shooting and star generation for the background.

Enemy_path

```
module enemy_path (input logic [7:0] time_passed,
                   output logic [9:0] X_Step,Y_Step);
```

Description/Purpose: This module controls the enemy path based on the amount of time that has passed. We need this module to control the movement of enemies because part of the objective is to destroy all the enemies

Time_ctr

```
module time_ctr #( parameter period =60)

              (input logic frame_clk, frame_clk_re,
                input logic clk,
                input logic count_on,
              input logic reset,
              output logic new_period,
              output logic [7:0] time_passed);
```

Description/Purpose: This module is a counter that keeps track of the time that has passed in the game. This module is by default parameterized to count the number of seconds, but we additionally use a quarter second version for enemy movement control and a frame counter version for seeding. We need to keep track of time so the user knows how long they have been playing and it also plays a part in calculating the final score.

Enemy

```
module  enemy ( input logic    Clk,                //
                               Reset,              // Ac
                               frame_clk, frame_clk_re,
                input [9:0]    DrawX, DrawY,       // Cu
                input [9:0] X_Step, Y_Step,
                input logic [3:0] enemy_type,
                input  logic [9:0]  InitX, InitY,
                output logic [3:0] lives,
                output logic [9:0] num_hits,
                output logic  is_enemy,
                output logic [9:0] X_ctr, Y_ctr,
                output logic is_enemy_bullet,
                input logic [3:0] level,
                input logic shoot,
                input logic is_ship,
                input logic is_bullet,
                output logic too_far_down
```

Description/Purpose: This module controls the specific instance of an enemy. It has an FSM that determines the state of the enemy: ALIVE or DEAD. It has hit detection and automatically transitions from ALIVE to DEAD based on the number of lives it has (updates internally). This module also control an instance of a bullet and shoots it in order to implement enemy shooting.

Bullet

```
module  bullet ( input logic        Clk,
                                    Reset,
                                    frame_clk, frame_clk_re,
                 input [9:0]   DrawX, DrawY,
                 input  logic [9:0]  InitX, InitY,
                 input logic shoot_key,
                 input logic [4:0] bullet_speed,
                 input logic enemy_bullet,
                 input logic  is_enemy,
                 output logic  is_bullet
                );
```

Description/Purpose: Each alive instance (enemy/ ship) has their own bullet module they can control. It contains an FSM that when the controller tells it to shoot a bullet, it will keep moving until it goes out of bounds or hits an object. We need this module to implement the shooting mechanism of the game be able to shoot and destroy enemies, and be able to get hit by enemies.

VGA controller

```
module  VGA_controller (input               Clk,
                                            Reset,
                        output logic        VGA_HS,
                                            VGA_VS,
                        input               VGA_CLK,
                        output logic        VGA_BLANK_N,
                                            VGA_SYNC_N,

                        output logic [9:0]  DrawX,
                                            DrawY
                        );
```

Description/Purpose: This module handles the actual behavior of choosing which specific part of the screen is currently being drawn. This is in raster-order: meaning that you move left to right in a row and when you reach the end of the row you go to the next row down in the first column. When you reach the last pixel (bottom right) you reset the position of where you are drawing to the top right. We need this module in addition to the outputs it generates such as VS, HS in

Vga_clk

```
module vga_clk (
    inclk0,
    c0);
```

Description/Purpose: This module generates a clock specifically for the VGA controller based on the system clock as an input. We need this clock to make sure we have the correct timing for the VGA protocol.

Lab8

```
module lab8( input               CLOCK_50,
             input        [3:0]  KEY,
             output logic [6:0]  HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7,
             // VGA Interface
             output logic [7:0]  VGA_R,         //VGA Red
                                 VGA_G,         //VGA Green
                                 VGA_B,         //VGA Blue
            input logic [17:0]   SW,
             output logic        VGA_CLK,       //VGA Clock
                                 VGA_SYNC_N,    //VGA Sync signal
                                 VGA_BLANK_N,   //VGA Blank signal
                                 VGA_VS,        //VGA virtical sync signal
                                 VGA_HS,        //VGA horizontal sync signal

               output logic [17:0]LEDR,
               output logic [8:0] LEDG
                    );
```

Description/Purpose: This is the top level: any module that will be synthesized on the board needs to be declared here. In our top level we have all our interconnections between the modules and the inputs/ outputs of this module are mapped to specific pins on the board in the Pin Planner.

Hex Driver

```
module HexDriver (input  [4:0]  In0,
                  output logic [6:0]  Out0);
```

Description/Purpose: The HexDriver was provided to us during Experiment 4 and was simply recycled for this experiment. The HexDriver translates a binary input representing a number into the specific segments of a 7-segment display that would need to be able to display that number in hex. This is used to display inputs on the FPGA board in run-time for debugging.

State Diagrams

Game Control

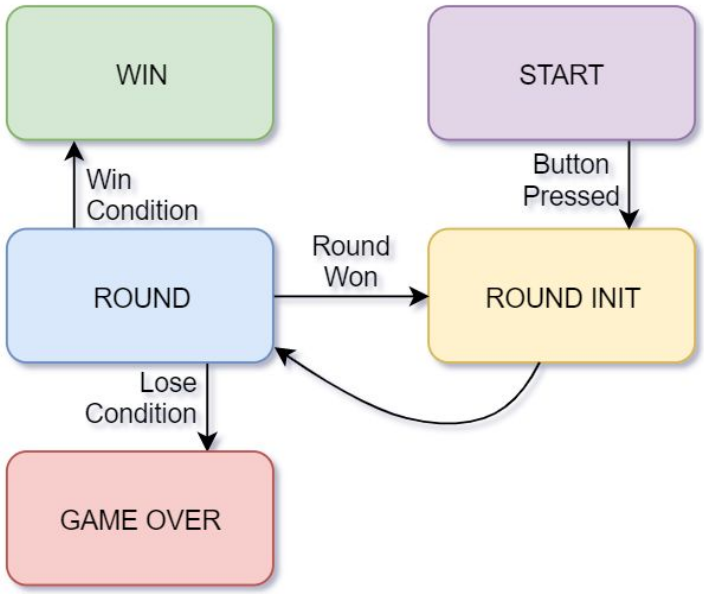


Figure 4. Game Control Finite State Machine.

On system initialization, the FSM will start in the START state until the first button is pressed. When the first button is pressed, the round number will be set to 1 and the FSM will send control signals to the rest of the circuit to initialize the first round. It takes an input, “Round Won”, which happens when either you destroy all the enemies or you manually override it with the next level button. In the ROUND INIT state it will increment the round number, send a control signal to initialize the enemies for the next round. There is an unconditional jump from ROUND INIT to ROUND. There is a win condition signal that is triggered when the round number is greater than 6, so after beating the last level we transition into the ROUND INIT state, increment the round number by 1, and after re-entering the ROUND state it will immediately jump to the WIN state since the round number is greater than 6. There is a lose condition signal that is triggered when it is indicated you ran out of lives or you let an enemy get to the bottom of the screen.

Enemy/ Spaceship

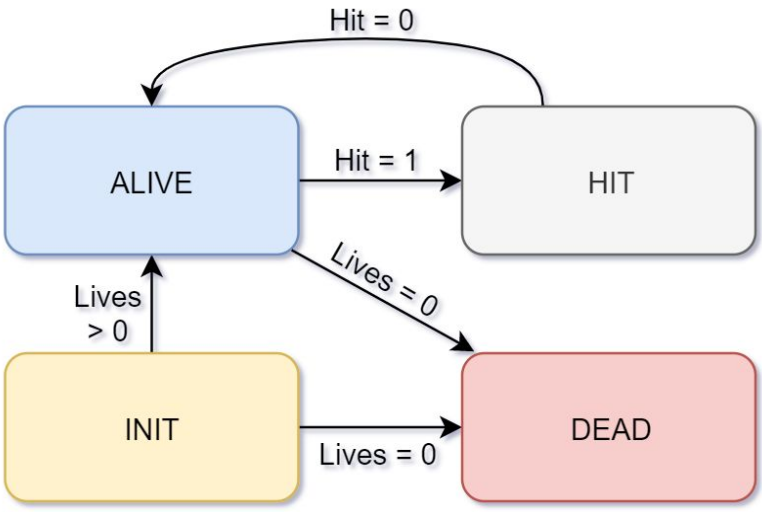


Figure 5. Enemy/ Spaceship Finite State Machine.

While the spaceship and enemies have different purposes, they have identical states and state transition logic. INIT is the default state after a reset. If that instance is initialized with zero lives it goes directly into the DEAD state (useful when we want to disable a specific enemy for a level configuration). Else, it will transition to the ALIVE state. The FSM has internal hit detection that will determine if there is a collision between the Enemy/Ship and the opposer's bullet, it will send a hit signal high. Since the object positions are updating at 60hz but the FSM runs at 50Mhz, we need to transition to a "HIT" state where we wait for that hit signal goes low again to avoid multiple collisions detected in a single event. The lives decrement by one and you go back to the ALIVE state. If the lives= 0 it will instantly transition to the DEAD state, else it will stay in ALIVE until another hit is detected.

Bullet / Star

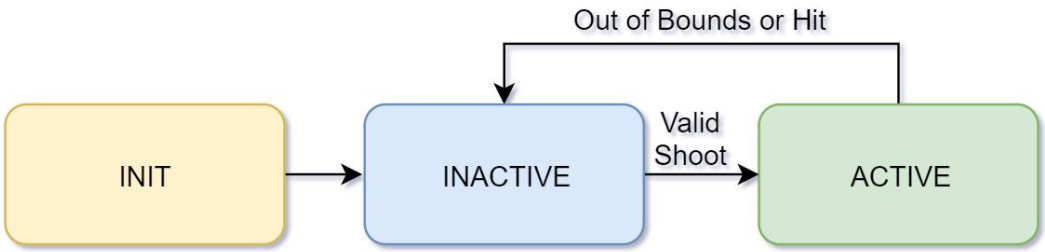


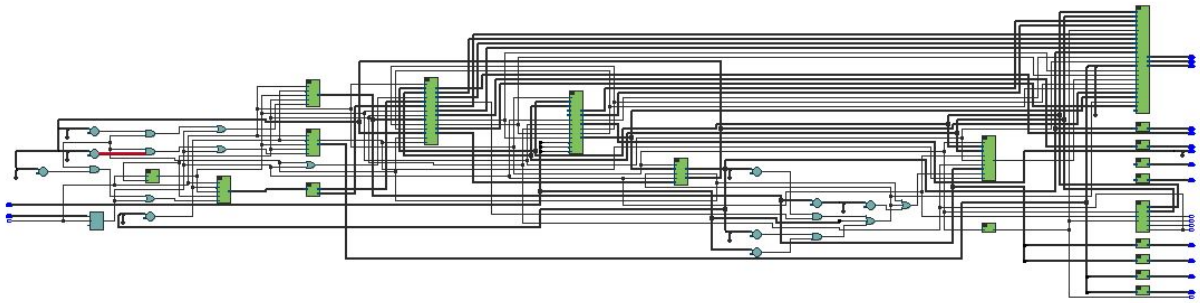
Figure 6. Bullet Finite State Machine.

For the bullet, this FSM is controlled by the ship/enemy, and for the star, the FSM is controlled is controlled by the shooting stars module. What this controlling module sends is a valid "Shoot" signal. Valid shoot is the case where a shoot signal goes high and enough time has passed from the previous shot. When that is the case it will transition and stay in ACTIVE as long as the bullet doesn't go out of bounds or hit an object. In the ACTIVE state the bullet simply moves vertically while in the INACTIVE state, the bullet position will constantly be updating to the controlling object's center but while it is inactive the bullet will be disabled for hit detection and will not be drawn. INIT is the default state after a reset and there is an unconditional transition from INIT to INACTIVE.

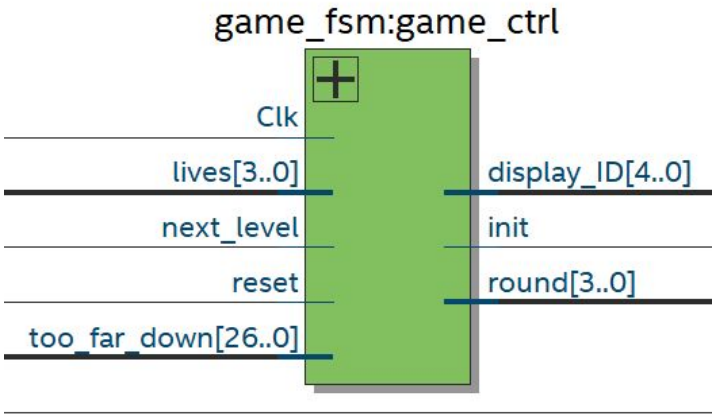
The Star has an identical state machine except there is no hit condition.

Block Diagram

Top Level View

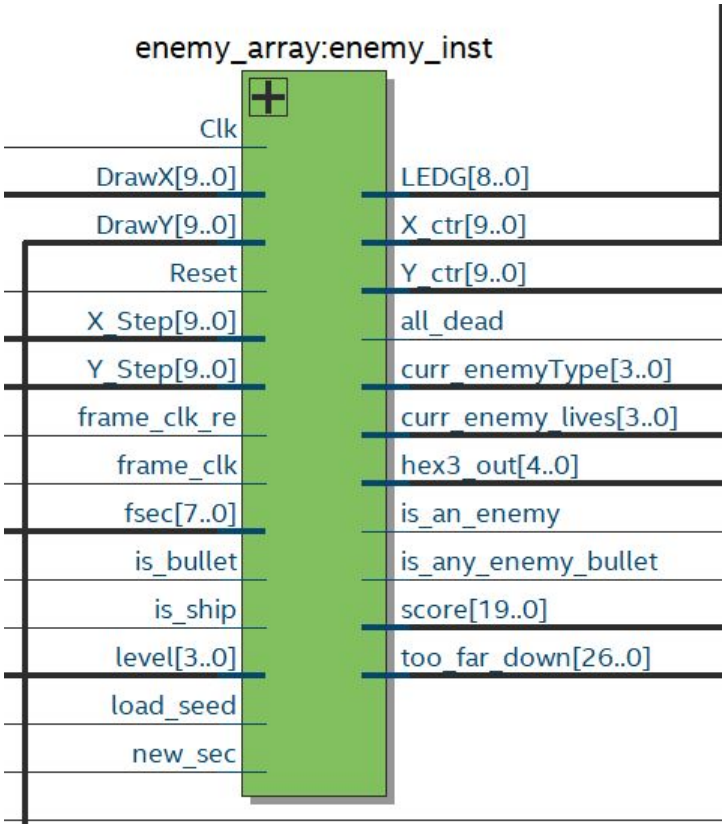


Game FSM

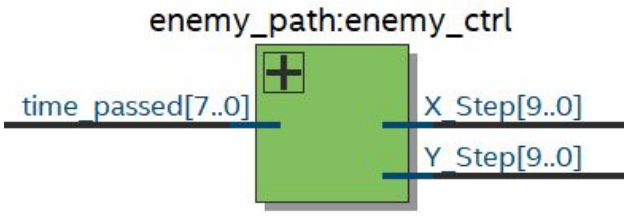


Enemy

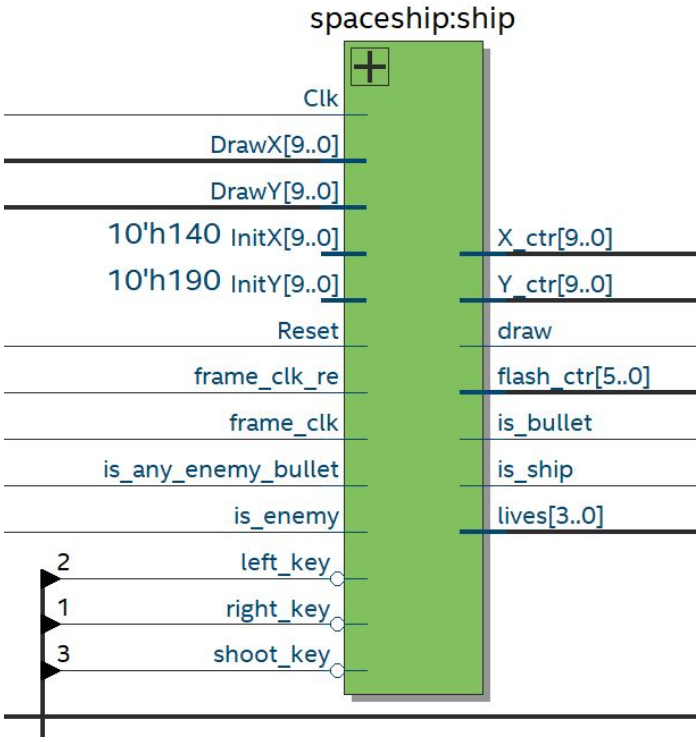
Enemy Array



Enemy Path

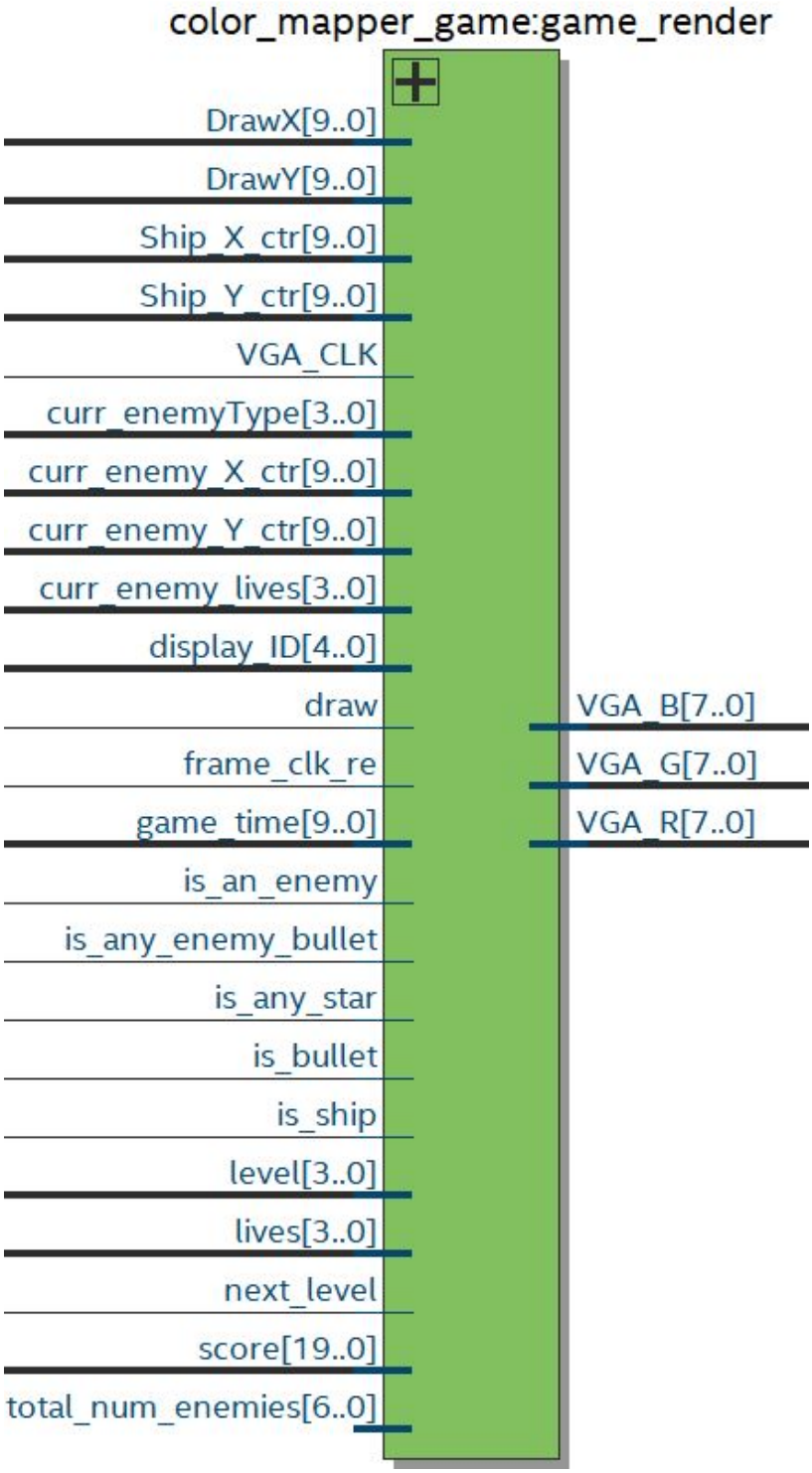


Spaceship

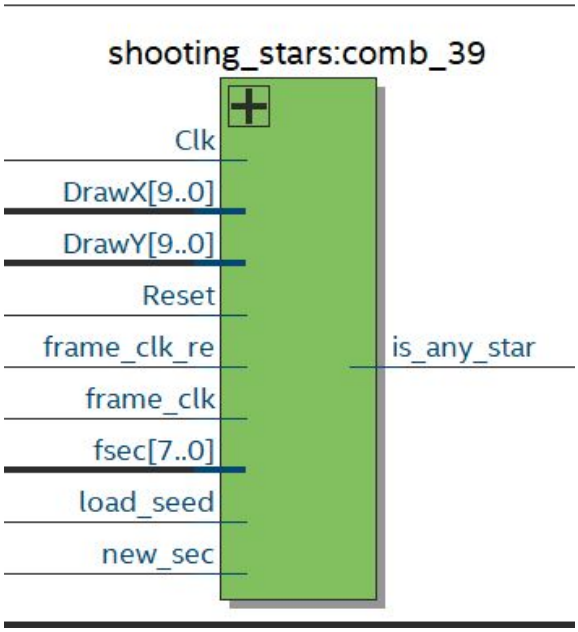

spaceship:ship
Clk
DrawX[9..0]
DrawY[9..0]
10'h140 InitX[9..0]
10'h190 InitY[9..0]
Reset
frame_clk_re
frame_clk
is_any_enemy_bullet
is_enemy
2
left_key
1
right_key
3
shoot_key
X_ctr[9..0]
Y_ctr[9..0]
draw
flash_ctr[5..0]
is_bullet
is_ship
lives[3..0]

Graphics

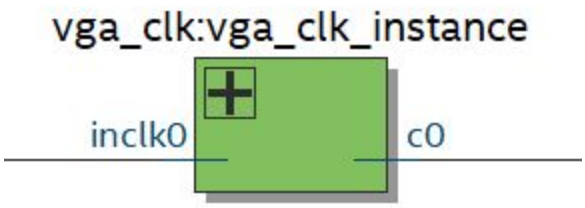
Color Mapper



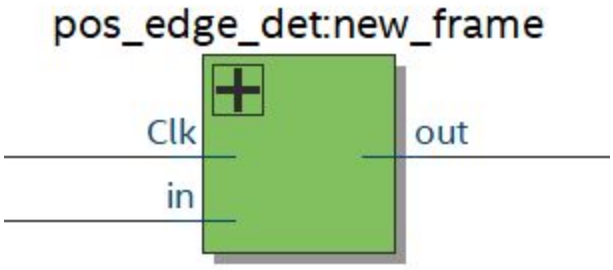
Shooting Stars



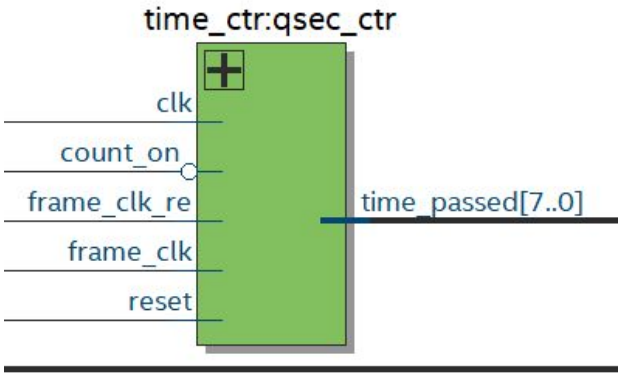
Vga Clock Instance

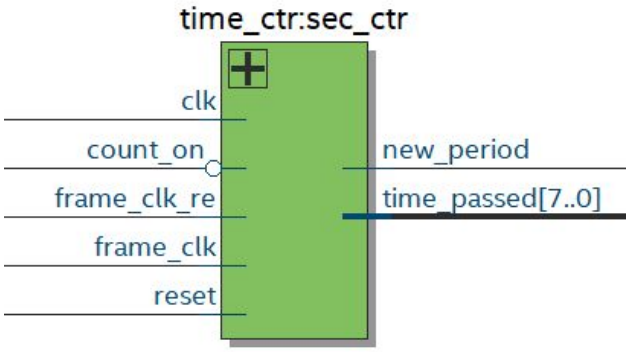


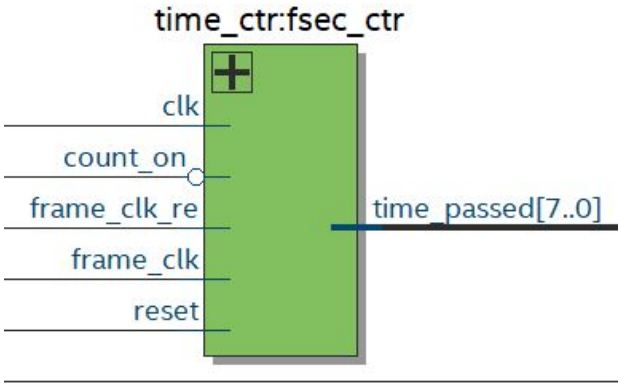
Misc



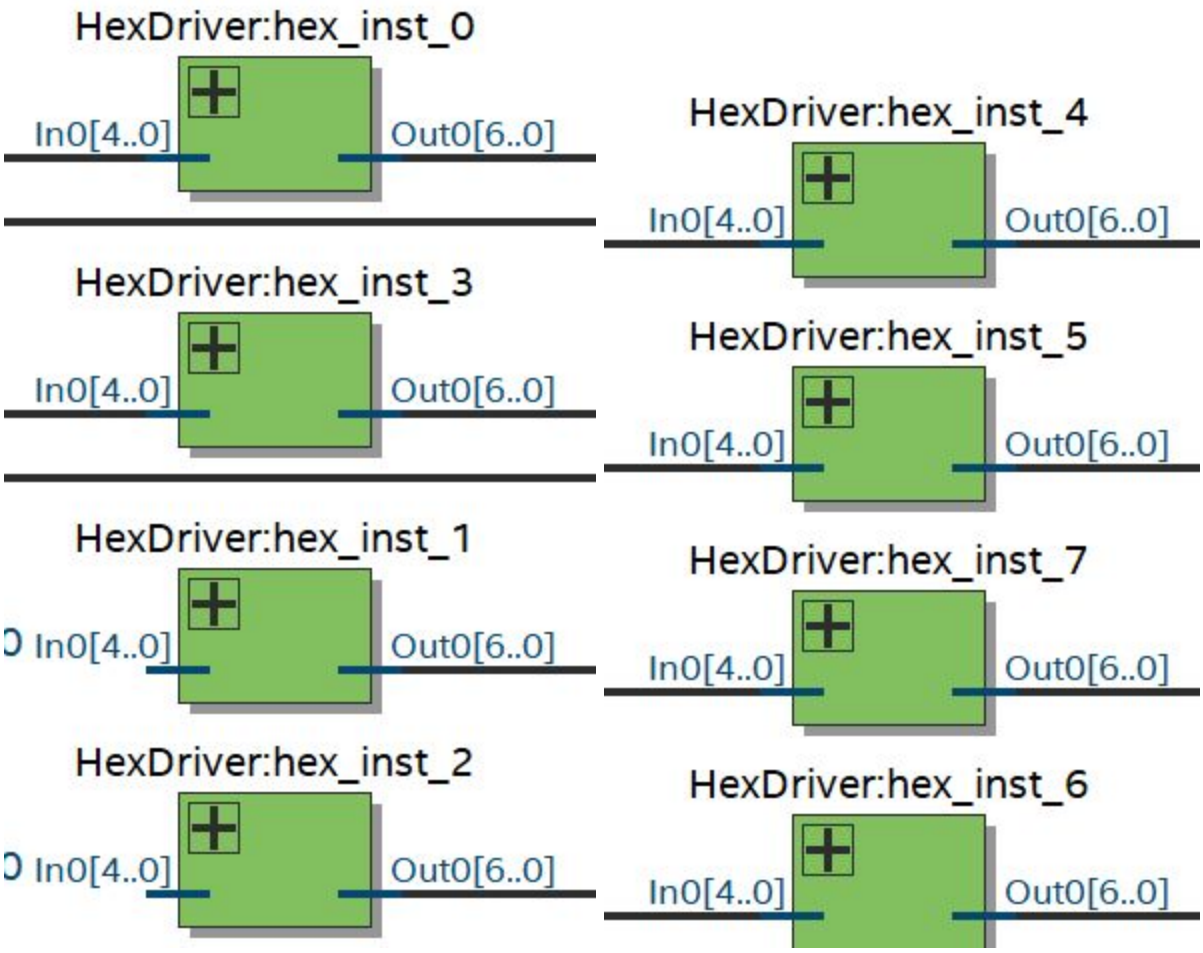
Time Counters







Hex Drivers



Design Resources and Statistics Table

LUT	**20,029**

DSP	**0**
BRAM	**0**
Flip-Flop	**1725**
Frequency	**78.51 Mhz**
Static Power	**103.02mW**
Dynamic Power	**0mW**
Total Power	**186mW**

Conclusion

In conclusion this final project encapsulated the topics and skills we learned after lab #3 alongside some material that we had to learn on our own or with the resources provided to us by the course staff. Some of the key topics and content that we used in our project multiple FSMs. We had FSMs controlling other FSMs. We also used the content learned through lab 8 to and applied it to this project, which allowed us to display content using VGA. However, we used no C code since NIOS since we didn't need anything that used a processor. We were able to implement everything using state machine and sequential/combinational logic. If we did want to use the NIOS, we had to synthesize it and also store the program, thus requiring us to use the platform designer and unnecessary complexity in terms of memory configurations. Also it was nice since hardware is faster than software. One topic that was not specifically applied in previous labs was the use of complex sprites and color palettes. However with the help of KTTech and some other resources provided for us, we were able to learn and apply them at a much greater ease. Overall, this project was a great learning experience and a way to express our creative side as well.