6.111 Project Report

Space Invader with a Twist

Tuan Phan

Edwin Africano

Table of Content:

1. Introduction……...…..3
   1.1. External Components……3
   1.2. Goals……3
   1.3. Results/Integration……4

2. Design Scheme……...4
   2.1. High-Level Block Diagram……..……..…4
      2.1.1. Data Flow……....4
   2.2. The Game Block Diagram……..……..5

3. Shooting Mechanism……..6
   3.1. NTSC Camera/Tracking (Tuan)……6
   3.2. Position Transformation (Tuan)……..……7

4. Controller…… 7
   4.1. SPI Protocol/Reading Data (Tuan).....8
   4.2. Data Calculation (Tuan).....9

5. Sound Effects/In Game Music.....11
   5.1. 8-bit Classic Music (Tuan).....11

6. Image Displays.....11
   6.1. Characters Display (Tuan).....11

7. Game Module(Edwin).....12
   7.1. Introduction.....12
   7.2. Enemy Control Module.....12
   7.3. Enemy Waves.....12
      7.3.1. Top Level.....12
      7.3.2. Enemy Instances.....15

7.3.3. Projectile Handler....15
7.3.4. Boss Health Bar....16
7.3.5. Miscellaneous....16
7.4. Shot Handler Module....16
7.5. Ship Handler....17
7.6. Divider....18
7.7. Convert_bcd....18
7.8. Score_num....19
7.9. Miscellaneous....19
7.9.1. Other Display Objects....19
7.9.2. Music Management and Score....19
7.9.3. SPI Modules....20

8. Menu Module(Edwin)....20
8.1. Introduction....20
8.2. Ship Module Instances....21
8.3. Star Module Instances....21
8.4. Picture Blob Instances....22
8.5. Support Logic....22

9. Assembly....22

10. Conclusion....22

11. Source Code……..……..23

1 Introduction

Electronic entertainment is rapidly moving toward a new future where virtual reality technologies will be in the spotlight. Our group's objective was to bring back the nostalgia of arcade games and give it a modern twist that would not alienate the younger players. For our project, we decided to create a game in the spirit of *Space Invader* coupled with shooting mechanics from *Duck Hunt*. Our shooting mechanism makes use of a laser and NTSC camera as the "gun" and an IMU as the gamepad. In order to achieve a playable game in hardware, many considerations had to be taken into account with the FPGA, namely the risk of timing constraints, memory, and computational cycle delays.

1.1 External Components

- NTSC camera interfacing with the FPGA producing VGA output to the screen.
- Accelerometer from the IMU (MPU-9250) to produce the controller for the game.
- 74LS00 chip to drive an external speaker for the sound effects.
- External speakers for the in-game music.

1.2 Goals

- Basic Goals
    - The ability to track a green laser projected on the screen.
    - A functional IMU to control the ship's speed and position on screen.
    - Created enemies on screen can fire projectiles.
    - Enemies regeneration by making use of the FSM.
- Main Goals
    - Use the camera and the laser pointer to implement the shooting mechanic by translating the position of the laser to a position in the gameplay.
    - Enemies organized into waves that have different enemy configurations and paths. The speed of the enemies scales with the score of the player.
    - A tone generator to produce sound effects for each different mode of the game.
    - A main menu screen and a game over screen to determine the start and the end of the game.
- Stretch Goals
    - Complex enemy behavior that depends on ship position.
    - Boss fight encounter.
    - Classic 8-bit music.

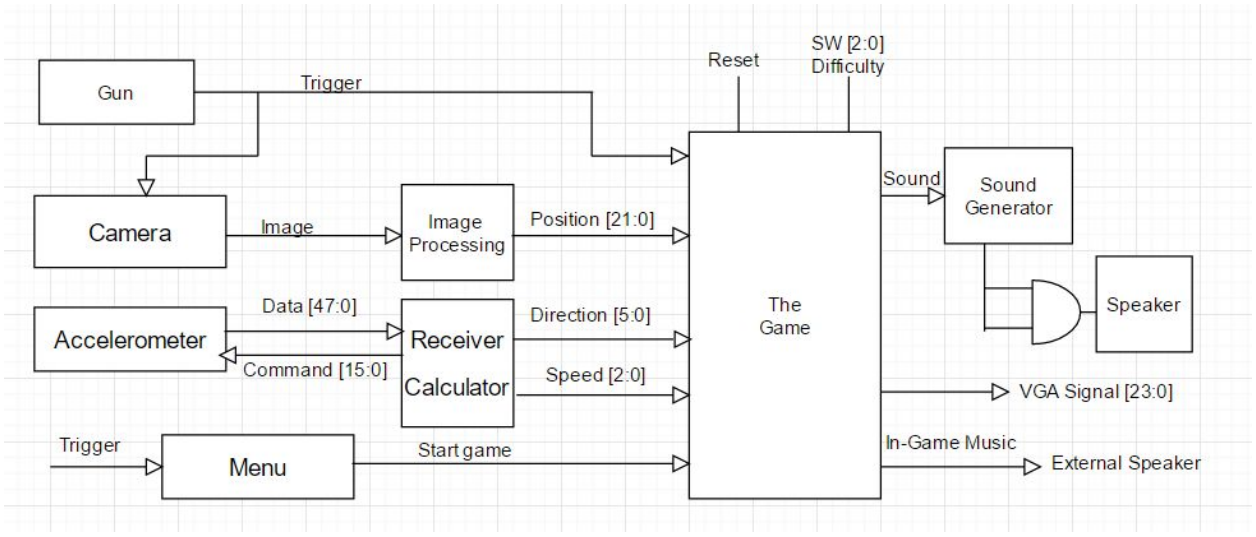
- Complex menu (difficulty options, ship model to use).
- More gameplay choices (ship types, powerups, etc).

1.3 Results/Integration

The project manages to complete all of our set goals although many of the functionalities could be further improved or refined. Since our project can be divided modularly, each one of us worked on our components, testing individual modules separately. When everything was assembled together, we encountered some interesting problems. Our display modules, such as the enemy sprites, displayed somewhat glitchy and did not look how they were intended. The camera's tracking is slowed down and sometimes did not function as well as it could. After talking to Gim, he suggested we should pipeline our modules in order to increase the system's throughput. Pipelining improved a lot of our sprite displays and made the game look a lot nicer. However, the issue of the camera's still appears in our project to a lesser degree. If someone wishes to replicate our project, this issue should be tackled first, the use of a larger FPGA would likely be enough.

2 Design Scheme

2.1 High Level Block Diagram



2.1.1 Data Flow

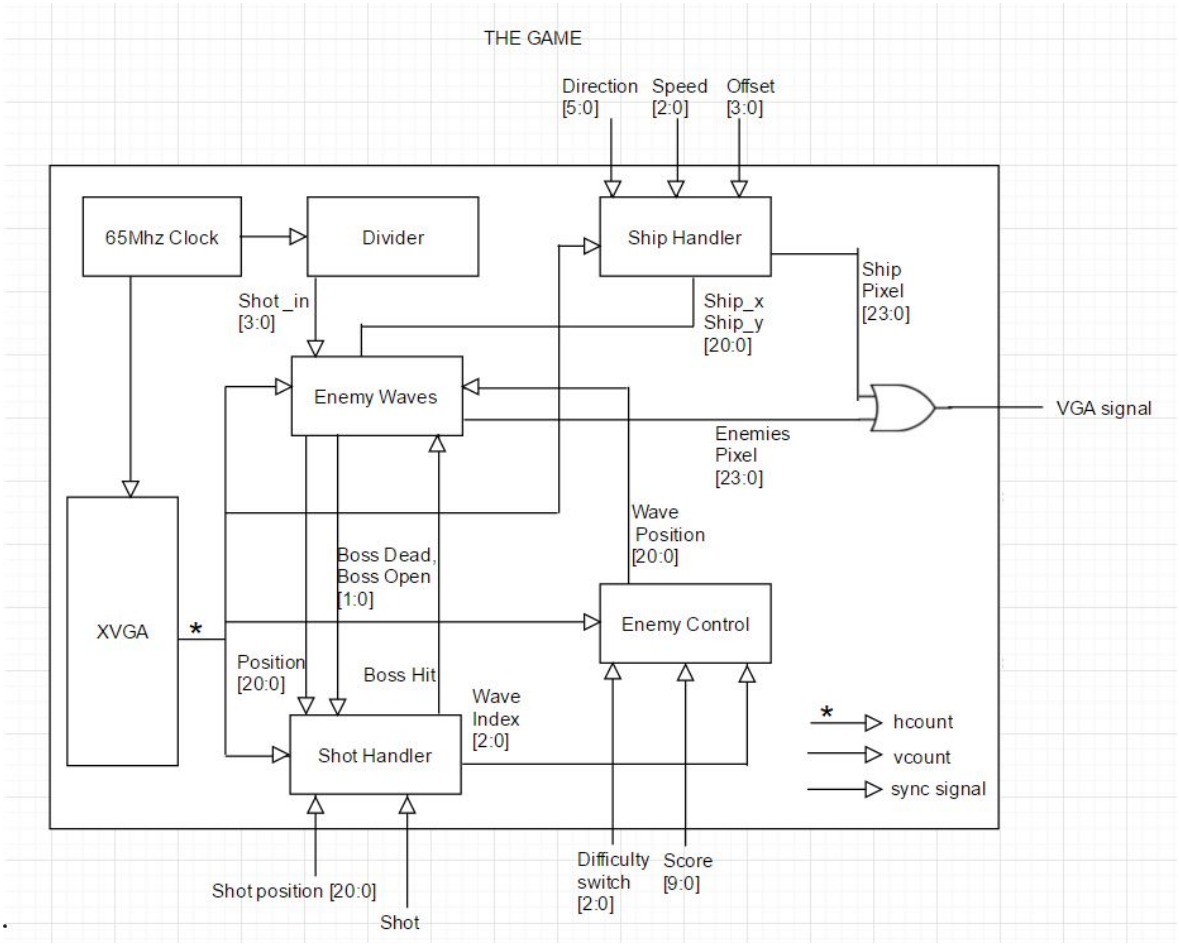
In our scheme, the "gun" includes a green laser pointer and an active low trigger button. When the button is pressed, our game then recognize the position of the laser

pointer on screen. Along with the laser pointer, the camera completes the game's shooting mechanism. The image processing block converts the black and white images to color images then filters out every color beside green for tracking purposes. In the image processing module, there is also a transformation step to map the camera position to a position in the 1024x768 VGA display. The transformation then outputs the calculated position to the game.

The game used an accelerometer for the controller and the accelerometer is sourced from the MPU-9250 chip. The data then goes to a receiver block which calculates the incoming data and turns them into useful information for the game. The receiver outputs direction and speed then feed the commands to the game.

In the project, we used a sound generator module to generate sound effects for different parts of the game. The game also utilizes the ac97 chip inside the labkit to produce in game music to give the users a fuller experience playing the game.

2.2 The Game Block Diagram



3 Shooting Mechanism

Originally we intended the shooting mechanism to be similar to the game *Duck Hunt* but through a variety of technical problems such as unidentified parts and constraints in the type of display, Gim gave us the idea of using the NTSC camera to track a laser pointer which would be the “gun” in our project.

3.1 NTSC Camera/Tracking

In our setup, the NTSC camera positions in front of the computer screen taking information of the display of the game. The camera outputs to the labkit analog signals with informations about the color, intensity, and synchronicity. The analog signals then get converted to digital signals with an Analog Device ADV7185. Originally, In the staff-provided code, the YCrCb components from the camera are stored in the ZBT Memory and then get displayed onto the screen. This resulted in a black and white display which would not allow us to perform a color tracking functionality. In order to perform tracking of a specific color, the YCrCb must be converted to color. From lab three, the YCbCr and RGB are related by this expression:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Using the staff-provided code (ycrcb2rgb.v), we converted the YCrCb to RGB signal allowing us to display colors on screen. The RGB values then feed into a color filter module, (colorFilter.v), to check for a green spot on the screen and turn the green pixels to a pure blue with the value of 256. The other colors turn black to reduce the distortions caused by other blue objects in game. The filtered RGB value then feeds into a center detection module where we used to calculate the center of the green spot on screen (center_detection.v). The module looks for all the green pixels the camera sees and then takes the pixel’s locations on the screen using the hcount and the vcount from the XVGA signal. The x and the y coordinates are then accumulated in a register. At the end of the frame, the averages of the x and y coordinates are calculated. This calculated averages provided a coordinate in the camera’s display which we can then use for the transformation process.

We encountered several challenges when implementing the tracking and transformation scheme. For the coloring filtering process, we initially converted the RGB values to HSV values. HSV values allow for a more robust system for tracking multiple colors at the same time, however since we ended up tracking only a single color for the system, HSV would only end up creating noises due to the other colored objects within the game. After several tests, we ended up using RGB filtering as it provided a more accurate tracking when displaying multicolor game objects.

For the center tracking process, after receiving the filtered RGB values, we store them in the ZBT memory which acts as a buffer to display onto the screen. To calculate the center of the light, we must use the data pass into the ZBT so that the clocks between the data and the XVGA data are synchronized. To detect the center of the spot more accurately, we must also eliminate noises from the corner of the images. In order to reduce the noise level from the camera, a “box” was used. The center detection module excludes pixels that are fifteen pixels from the borders and this scheme reduces the noise level drastically allowing the pointer to be tracked more accurately.

3.2 Position Transformation

Unlike the VGA display, the camera signal does not have a 1024x768 resolution. Therefore, a transformation module, (grid_lookup.v), maps the location of the pointer on the camera’s resolution to the 1024x768 VGA display. Initially, the module would take in four corners of the screen then map the pointer position in relation to the four corners. However, this process would take a lot of calculation cycles and would not be quick enough for a game. In order to resolve this issue, the camera in our setup is looking directly at the screen so that the transformation is linear eliminating the delay that would required otherwise. In order to have a more accurate pointer’s location in game, we also have a calibration screen that have four red boxes. The transformation module splits the camera display into grids of equal size. A grid would be an equivalent to a 16x16 grid on the XVGA display with the resolution of 1024x768. Using this scheme the transformation module would provide the coordinate of the top left corner of the 16x16 grid. The coordinate is then feed to the game whenever a button on the controller is pressed.

4 Controller

For the game’s controller, the goal is to mimic a gamepad on a console and add some extra functionality. The controller uses an IMU chip, MPU-9250, to perform its function. An added button was included to act as a trigger for the laser pointer.

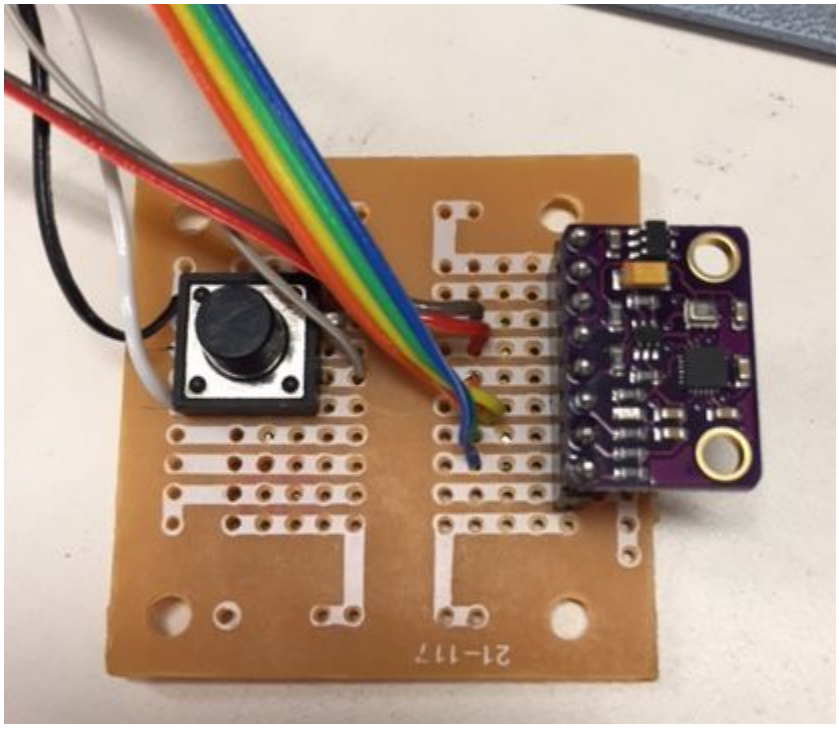


Figure 4.1. The controller

4.1 SPI Protocol/Reading Data

To communicate with the IMU, a SPI protocol was implemented (spi_reading.v). Using an SPI protocol is advantageous in the project because communicating with the chip using SPI protocol is easy to understand and the speed to sample data is quite fast, up to 10 Mhz sample rate. SPI also uses a minimal number of wires, four, to communicate compared to other methods (only one more than I^2C protocol). SPI communication is also relatively universal therefore many chips that use SPI protocol can be implemented using the same code and the same setup.

In the SPI protocol, there are four wires used to communicate the IMU with the FPGA: a clock between the chip and the FPGA (sck), a chip select, a master-in-slave-out wire (miso), and a master-out-slave-out (mosi). In this setup, the IMU is the slave in the system and the FPGA is the master. The master generates the sck clock and this is one of the factor that determines the sample rate of the system. Since the IMU has an accelerometer, a gyroscope, and a magnetometer, a chip select is needed to enable which device is needed for the application. The miso wire is where the master, the FPGA, receive the data from the chip. The mosi wire is where the master sends a specific command to the chip which can be used to manipulate the

threshold of the chip, to interrupt data when specific data is received, to select which register to read from, etc.

For the protocol in the idle state, the chip select wire, the sck clock, and the mosi wires are high, ready for changes. In the protocol, the master must send to the slave a 16-bit command. The most significant bit is to indicate whether to master wants to read from the slave or write to the slave. The next seven bits in the sequence are to indicate what register the master will read from or write to. The least eight significant bits are used to write to the slave to indicate what needs to be changed.

When starting to recieve data, sck clock and cs must then pulse low. This is an indication that the master is ready to send out commands to the slave. This must always occurs at every reading because if the chip select wire remains low and sck clock continues to run, the IMU would not know what commands is being sent to it. A change in the sck clock and the chip select wire acts like a wake up call for the chip to start working. After the eight most significant bits are sent, if the first bit indicates the command to read, the miso wire then received eight bits, from most significant bit to least significant bit, of information from a specified register. When reading the accelerometer, there are readings from three axes (x,y, and z) and each axis contains a 16-bit two's complement data, which means the readings can contain negative values and have a total of 48 bits. The protocol also makes this retrieving data process simpler by allowing the user to read all 48 bits without the need to specify a new register. In order to do this, the sck clock just need to remains running after the first eight bits and when the user wants to stop reading the data, the user then needs to return the sck wire and the chip select to the original idle state. The majority of IMU chips have the registers for all three axes reading right next to one another.

4.2 Data Calculation

After receiving the 48 bits of data from the accelerometer, the bits then can be separated into three 16-bit two's complements. Since the data is in the two's complement format, the user can manipulate data using basic arithmetics. Using these readings, this module (spi_calc.v) translates the data to useful information for the game such as the direction and the speed of the ship.

Initially, if the user just reads the data directly, there would be a lot of noise in the system due to the mechanical aspects, such as the spring and mass constants, and the electrical thermal noises of the system. In order to reduce noises, over time averages must be used. In the project, the calculation module takes in eight different readings and then take the averages for each axis. Furthermore, the rate of data the module receiving is approximately 500 Hz. Using this sample rate, the noises of the IMU are reduced

significantly giving more sensible data. After some testing and approximation, the user can then see what value should be used for a threshold for whatever direction or orientation the IMU is being held at.

In the scheme for the game, the threshold is set at half the earth gravitational accelerations. If the tilt exceed this threshold in the x and y axes, the ship will move accordingly to the tilt direction. Two interesting problems came up when setting the threshold of the IMU. The first one is the direction of the chip. In the data sheet, the axes of the MPU-9250 are clearly labeled and are very conventional with how one would draw a three axes graph (Figure 4.2). But during the testing process, the axes are completely opposite with the datasheet. Therefore adjustment of the thresholding values accordingly to the new directions is necessary.

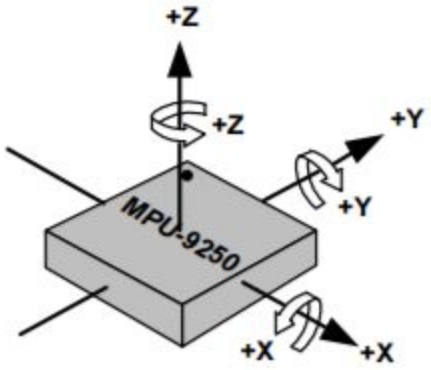


Figure 4.2. Orientation of Axes for Accelerometer Provided on Datasheet

Another problem that came up is jittering of the image even though the IMU is stable and laying flat on the table. The jittering occurs due to the x and y axes' readings are alternating between a small positive and a small negative number. For two's complement values, a small negative is an actually big number in binary term. For example, -2 in 16-bit two's complement is 16'b1111_1111_1111_1110. Sometimes, the alternating between these two numbers would become a big number exceeding the threshold leading to the jittering of motion. In order to solve this problem, a feedback system was used that would weight a previous average and compare it to the new average. The implemented scheme was: $y[n] = 0.95 * y[n-1] + 0.15 * x[n]$. By using this scheme, the jittering was eliminated almost entirely.

After the implementation of the direction of the speed, the game also called for a way for the user to change speed of the ship. The scheme for the speed was quite simple. Since the default limit of the accelerometer is $\pm 2g$ which is around 32,000 in

binary, the limit of the controller rarely reachs $1g$, which is around 16,000, unless the side of the IMU is completely parallel with the ground. By setting the threshold at 8,000, every 2,000 incrementation from the 8,000 point increases the speed of the ship by one.

5 Sound Effects/In-Game Music

To make the game more exciting and similar to most of video games, music and sound effects were implemented (sound_generator.v). For the sound effects' set up, a NAND gate, 74LS00 chip, was used as the driver for the speaker. The input into the speaker was various tones from 400 to 700 Hertz with various combination to generate interesting sounds and noises. For the game, the module generates sound at the start of the game, each time the enemy shoot, and game over.

5.1 In-Game Music

For the music during the game, a classic eight bit sounding arcade music was used for the system. The song is originally sampled at 44.1 kHz. In order to make the music sound good, the system must be able to play at least 15 seconds. If the original sampling rate was kept, the memory used was approximately 5280k bits and the labkit only posses 2952k bits in total. To save memory inside the labkit for other modules, the song was downsampled to 8 kHz. With 8 kHz sampling rate, fifteen seconds of music would cost 960k bits, substantially less than the original sample rate. The data is then sent to the AC97 chip. Using the audio lab's scheme without the recording functionality is sufficient for our purpose (music_player.v).

6 Image Displays

In the interests of making the game look better and incorporate a complex menu with an option to choose a ship type, displays were needed.

6.1 Character Displays

Even though the displays are characters, they are actually images created using Paint. The images are then turned into bitmap and using MATLAB, the images are turned into black and white. Once the bitmaps were black and white, the images are then converted into COE files which can be stored in a ROM on the labkit. The module that is responsible for the displays then checks each of the binary value in the COE file and if the value is zero, the pixel is then displayed on screen otherwise the pixel is black to mix in with the background. The game has three character displays: choose ship, start the game, and game over.

7 Game Module

7.1 Introduction

The game module is in charge of instantiating all of the elements that comprise the video signal generation for the game. This includes the sprites for the enemy's , player ship, score, remaining lives, and projectiles. It also includes the modules in charge of generating the correct enemy behavior for each enemy wave. Finally it includes all the instantiations of Tuan’s audio module, SPI Controller module, and clock divider modules. Besides the instantiation and connection of these modules, we also have some logic that acts as an FSM to handle score and sound output, as well as video muxing for the XVGA signal.

7.2 Enemy Control Module

We wanted the incorporate an easy to use framework to create new waves as needed. This module is in charge of simplifying the generation of motion for all 16 enemies in the game. Making use of a case statement we use the (wave) signal to determine what wave behavior we should utilize. In general these can be thought of as sweeping up and down, moving left to right, and many others. This proves useful if we are only concerned with putting enemies in a formation and do not care about their individual movements. The default case is left as a value useful for generating static waves that have enemies follow uniquely defined paths.

The wave positions are outputted through the wave_x and wave_y ports.The module also calculates the difficulty of the game making use of the following lines:

```
x_speed <= 11’d2 + sw + (score>>.4);
y_speed <= x_speed;
```

This is a linear combination of our default speed , the (sw) signal which is set to the lab kits switch[4:2] , and the current game score divided by 16. The speed is only updated on a wave change, which ensures that the game always starts at the lowest difficulty and can ramp up on subsequent waves.

7.3 Enemy Waves Module

7.3.1 Top Level

This module is the next step in generating the enemy behavior for the game. In the Enemy Control module we generated a moving value stored in wave_x and wave_y. In this module we take the wave_x and wave_y modules and we pass a modified version of them to the individual enemy module instances.This is accomplished making use of two case statements. The first case statement is in charge of generating the enemy direction, this is done independently of the enemy position generation. The enemy direction are mapped as follows:

{0,1,2,3} = {up,down,right,left}

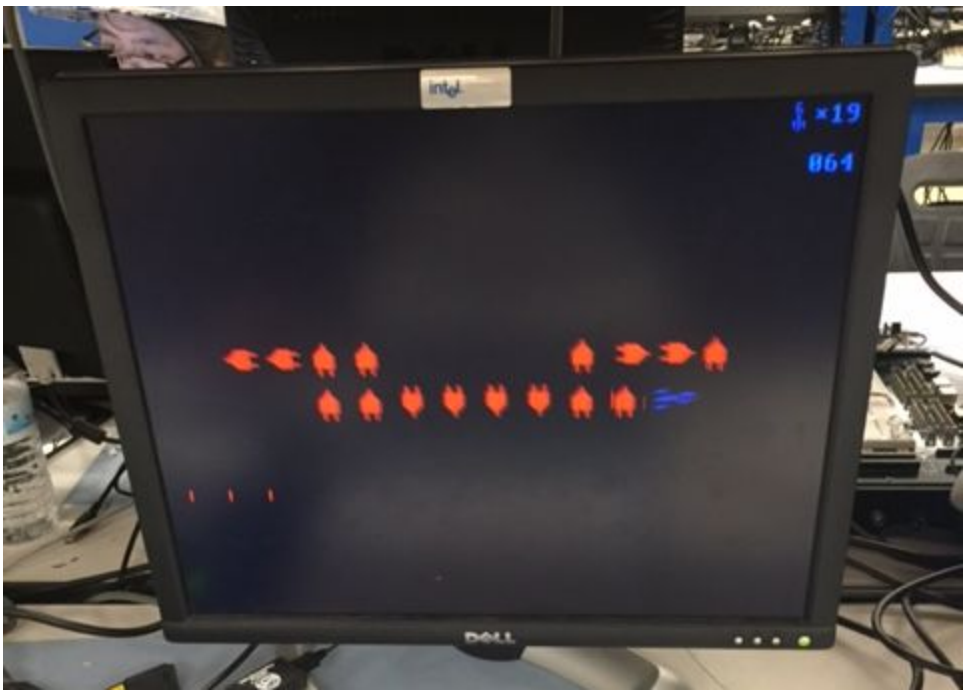


Fig.7.1 Enemy Wave: wave_x and wave_y update to move diagonally on the screen. All enemies in this wave are placed relative to these coordinates, note the direction does not directly translate to movement in that direction. All enemies move diagonally.

The game includes four types of enemies (AND,OR,XOR,BOSS) in order to specify the enemy type to be used we add an offset to the direction. The offset values are mapped as follows:

{0,4,8,12} ={AND,OR,XOR,BOSS_Legs}

Directions 16 and 17 are reserved for the boss center in both open and closed states and must be added manually if desired.

The second case statement is in charge of actually setting the final enemy behavior at the individual enemy level. Here we can take wave_x and wave_y and place an enemy relative to this point on the screen, their direction is determined by the aforementioned direction case. It is important to note that this does not need to be the case for each enemy. It is entirely up to the designer to choose whether to create custom behavior or utilize the wave_x and wave_y values. In waves 6 and 7 it can be observed that some enemies have their x and y values dependant on the ship position and their current position. Below are two examples of enemy position assignment:

```
{Rx_4,Rx_5,Rx_6,Rx_7}
<={wave_x+11'd200,11'd460,11'd524,wave_x+11'd700};

{Rx_5, Rx_6} <= (ship_x >= Rx_11)? {Rx_11 + 11'd1, Rx_11 + 11'd1}:
                                   {Rx_11 - 11'd1, Rx_11 - 11'd1 };
```

The upper expression shows 4 enemy x positions being set. Two are relative to the wave_x position while the other two are hardwired to a specific value. In the second example we have two enemy assignments. Neither assignment is related to the wave_x value instead they are relative to the ship_x position. This code will make sure that in the x axis these two enemies follow the player around. The condition is checked on Rx_11 as that is the center of the boss, it can be then deduced that these enemies are located above and below the center of the boss.

In addition to enemy movement this module is also in charge of instantiating the modules for the 16 enemies and their related projectile modules.The projectile modules are designed to follow the x and y position of the enemy modules when they are not firing.They can modify their location relative to them depending on their direction and also the one_hz fire mechanism.

In order to correctly determine whether a laser shot hits an enemy we require to know the position of all enemies. This module iterates over all enemies and outputs the (position) signal. This signal is composed of the enemy x,y position (Rx, Ry). the ID of the enemy, and whether it is alive or dead. As an input we receive the “killed_en” signal which is used to enable or disable the enemies as they are killed. This signal is 16 bits and each bit corresponds to an enemy and their associated projectile.

In the case of wave 7, the boss encounter, we have some added logic that ensure that the boss health bar is displayed and his health is only diminished when it is shot in the middle of its body when it core is exposed.

7.3.2 Enemy Instances

The enemies are individual instances of the module “enemy_and” which handles the display of a single enemy. The enemy_and module is found inside the sprite_video_modules.v file. This module takes as an input the position of the enemy, the video signal hcount and vcount values, a direction, and an enable signal. It outputs a 24 bit pixel value. In order to determine if the pixel should be black or display a color we do two things: first we check if hcount and vcount fall within the borders of our desired display space, if not the pixel is black, else we check if the particular pixel inside of our area has a color. This is done by addressing the instance of the enemy_rom. In order to correctly select which sprite to look for we feed the rom an address of 9 bits. The 5 more significant bits are our direction and they choose the block in the ROM to look at. The bottom 4 bits are used to identify the line in the sprite, and finally hcount is used to isolate the individual value at a point in the sprite. There is some additional logic that allows us to scale the size of the sprite, these are: size and scaling. The default values for these should be set to 2 and 64 respectively to generate a sprite 64 by 64 pixels. A value of 1 and 32 for example yields a sprite that is 32 by 32 pixels. Note that this scaling must be done in log base 2 in order for it to work correctly as there is no floating point scaling.

7.3.3 Projectile Handler

The projectile handler module (proj_h) is used as a wrapper of the enemy_proj module. The enemy_proj module works identically to the enemy_and module with the difference that it includes a different ROM module that holds the sprites for the projectiles. The handler is in charge of modifying the behavior of the projectile before it passes the values to the enemy_proj module instance. It takes whether there is a new enemy shot which happens at one_hz and if so the projectile moves in its current direction until it is offscreen or a new shot is fired. Otherwise the handler looks at the direction of the enemy and places the projectile in the correct position (above,below,left or right of the enemy) and disables the projectile pixel until a new shot is fired.
This is all accomplished making use of simple case statements that cover all possible directions (including the boss).

7.3.4 Boss Health Bar

This module instantiation works just like the blob module studied in lab 3. It replaces parameters with registers to modify its dimension in real time. It is off by default unless it's wave 7.

7.3.5 Miscellaneous

All output pixels from the enemy and the projectile modules are pipelined before they go through and OR gate to generate the final pixel output. There are also assign statements that correctly calculate internal enable signals dependant on designer intent.

7.4 Shot Handler Module

This module is in charge of handling every time the player fires on the screen. It takes as input the position of the enemy, the position of the shot, the shot trigger, and associated boss fight signals.It outputs an updated killed_en signal that controls the enables of individual enemies, the current wave number that all modules use, and whether the boss has been hit.

In order to handle a shot we wait for the (shot) signal goes high, thereafter the module begins to check whether the position of the current enemy matches the position of the shot and also if the enemy is dead or alive. Based on this check the appropriate bit of the 16 bit enable signal is updated. This process is repeated 16 times making use of the counter "i" , as we need to loop over all enemies for every shot in order to ensure that we handle all possible enemies. After 16 cycles have passed if the player does not let go of the button the module stops checking for hits to ensure the player does not "paint" the screen with his laser.

In the case that all the enemies have been defeated for a particular wave, the module soft resets and sets the enable signal bits all to high and increments the wave signal by 1. This serves as the transition between waves for the whole game. This transition is different for the 7th wave as this is the boss fight and the transition condition is set to the boss_dead signal. In this wave the boss is only hit when its core is exposed and the laser hits it. It's health is handled in the Enemy Wave module and fed back as the boss_dead signal.

Upon defeat the boss, the wave counter returns to 0 and the game begins loops over the waves again until the player looses all its lives or the reset signal goes high.

Fig 7.2 Boss Encounter. Here the boss has full health and its core is not exposed, the player is unable to damage it or any other enemies.

7.5 Ship Handler Module

The Ship Handler module is a wrapper that modifies the values going into an instance of the Ship module. The Ship module is nearly identical to the Projectile and Enemy_And modules, except that it uses the ship ROM instead.

The handle is in charge of taking in the heading_x and heading_y inputs from the SPI controller to change the direction of the ship. It also takes in from SPI the speed of the ship. In addition it can change the ship type by adding an offset to its direction mapped as follows:

{0,4,8} = { 6.111.6.101.INVERTER}

The module does the direction calculations with simple if/ else if/ else statement which also have hard constraints to ensure that the ship remains inside the screen at all times (i.e hits a wall on each one of the four sides of the screen). The handler also takes an input in the form of a collission_en signal that reset the ship position if it collides with an enemy or projectile.

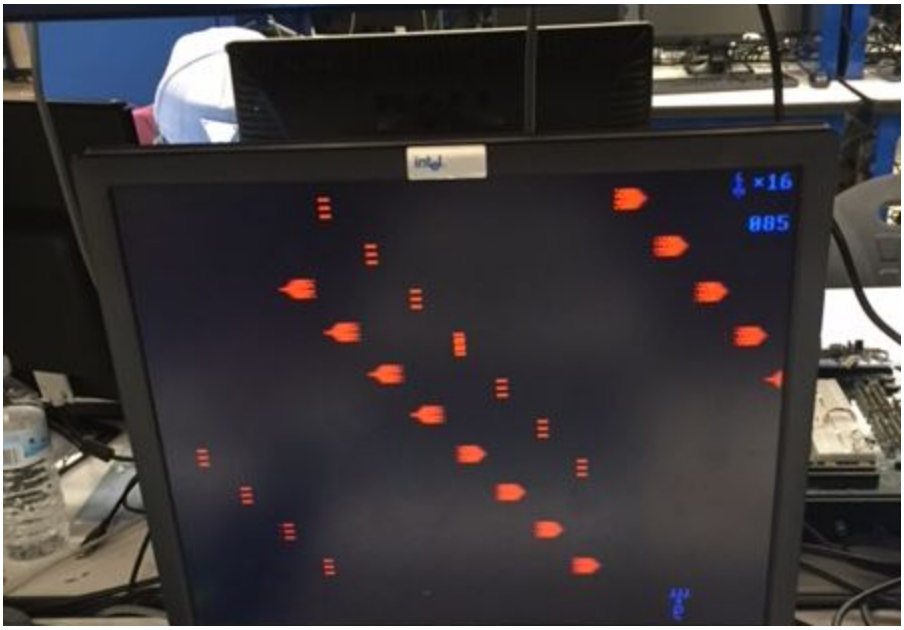


Fig 7.3 Ship at Edge. As can be seen the ship is unable to continue moving beyond the bottom boundary of the screen. The enemies do not have this limitation as they use the Enemy_and module without any handler.

7.6 Divider

This module generates a series of different enable signals with different periods to be used as toggles for other modules. It takes in our vclock and turns it into one_hz,two_hz, half_hz, and one_half_hz enables. These signals are used to do different firing rates for enemies and how often the score is updated. The module is just a slightly modified divider module from lab 4. It uses counters to create an edge only one time out of many cycles which creates a different period.

7.7 Convert_bcd

This module takes our 10 bit binary score and turns it into three signals corresponding to the ones, tens, and hundred components of the score value. It is identical to the one assigned in lpset_8.

7.8 Score_num

This module is another video display module identical in function to the Ship and Enemy_and modules. It utilizes a ROM that only contains sprites for numbers. Each of the three Dig_x instances display a component of the score value, making use of the score_con module outputs.

The other two instances of this module are used to display the components of the lives counter for the player. Their input value comes out from the second convert_bcd module “lives”, which uses the lives counter as its input.

7.9 Miscellaneous

7.9.1 Other Display Objects

In addition to the aforementioned digit, ship, enemies and projectiles (inside Enemy_Waves) , two other objects are also displayed: A scaled down version of the ship in the corner and an “X” sprite next to said ship. These are generated using a Ship module instance (no wrapper) and a custom Ship module called “X” which has a single sprite in its ROM and is otherwise identical.

7.9.2 Music Management and Score

The only logic besides the instantiation of all the aforementioned modules, their connections, and the pixel assignment ORing all the different pixel values is located inside the always block. Inside the block we have two separate sections, the first sets the values to be used by the instantiation of Tuan’s sound effects module. It does so by taking in an external signal intro_end (from Menu) and the internal collision signal. In this manner it can decide whether to play the startup sound, enemy projectile sound, or game over sound.

The second section inside this always block handles the calculation of the score and the update of the score display. The score register is updated when the Killed_en signal changes and is not zero, it gets incremented by 1. If Killed_en is zero the wave is dead and the player gets an extra 5 points. This section also handles the lives remaining and live display. We take the collision signal which is high when the ship pixel overlaps with a non black enemy or projectile pixel and use it to enable a collision_en signal. This signal going high signals a 1.5 second timer to ensure that the player gets a leniency period after each death. After this period is over the whole process can be repeated again. The collision_en signal also is used to decrement the collission_counter

which holds the number of lives remaining (initialized to 25) and whose value is passed to the dig_s0 and dig_s1 modules after passing through the lives covert_bcd module.

Fig 7.4 Score and Lives display. In the upper right corner we can see the remaining lives above the current player score. This is Wave 4 and the player Has lost 4 lives.

7.9.3 SPI Modules

Tuan's SPI modules in charge of feeding the ship module it's heading_x and heading_y values. They also generate the ship speed. They are instantiated inside this module for convenience.

8 Menu Module

8.1 Introduction

The Menu module is in charge of providing the user interface necessary to change settings in the game and from which to launch the game. It is mostly comprised of video modules for all its visual elements and some logic to generate functionality. Our Idea for the opening of the game was to have the ship travelling at faster than light speed and upon the player shooting the "Shoot to start" sign have the ship quickly accelerate off screen. Once the ship left the screen the game would replace the menu.

Additionally we wanted to allow the player to choose different ship models and was implemented in a similar fashion with the “Choose Ship” sign. There are three different models for the player to choose, once he chooses he is returned to the main menu and the ship on display is updated

Fig 8.1 Main Menu. All the menu elements are displayed. Upon firing at The “Shoot To Start” sign the ship accelerates and the screen switches to the Game.

8.2 Ship Module Instances

There are four Ship Module instances (No Handler). The first instance is the one on display at the start on power up. Its type is set by an offset variable that is only changed when the player selects a different ship. This offset is passed out to the game module. The three remaining ship instances are disabled until the player shoots at the “Choose Ship” sign at which point they are the only elements enabled. The player can then shoot at their sprites to select them for gameplay.

8.3 Star Module Instances

The star module works in the same manner as the Ship module, it differs only in the ROM used. The stars are placed at locations on the screen and their speed is

constantly incrementing (and resetting). While the area of each sprite is only 64 by 54 pixels, in moving their position using the vclock instead of waiting for the condition where hcount and vcount are at the bottom right corner of the screen, we cause them to move incredibly fast painting the whole screen with a cool effect.

8.4 Picture Blob Instance

This is an instance of Tuan's module as described in section 6.1. Here it is used to generate the "Shoot to Start", "Choose Ship", and "Game Over" images. The "Game Over" image is only displayed if the game has started and the player then loses all lives. For more details see section 6.1.

8.5 Support Logic

The main menu has a lot of support logic needed to generate the core functionality and sync it all together. All the logic is done inside the vclock always block. The counter register is used to ensure that the ship animation on startup syncs correctly with the sound coming out of the Sound Effect generator. Here also all pixel registers are pipelined for optimization.. All the following if statements are used to detect where the payer is shooting on screen and correctly update the state of the menu. Finally the vcount and hcount section is used to move the ship upon receiving the start intro command, it also signals the exterior of the game when the animation is over and the game should be brought to the foreground.

9 Assembly

The game, menu, and camera modules were all instantiated in the labkit.v file. The template is similar to the one used for lab 3. All appropriate wires are instantiated and connect the different components. A small amount of logic was added to switch between the Menu and Game pixels given the end of the intro sequence or the death of the player.

10 Conclusion

We believe that our game is a unique take on the old arcade shooters of the 70's and incorporates a fun and challenging shooting mechanic. We faced many challenges during the process but we also had a lot of fun creating the game. Even though almost all of our issues were resolved, if we get a chance to revisit the project again, we would make the game a lot more challenging and try to improve the player's experience with the game.

We would like to thank Gim, Joe and all of the TAs who pointed us in the right direction during the process of making the game. They gave us insights and solutions on many project's problems. We learned a lot about FPGA and digital system during the making of the project. Above all we hope that someone out there can get to enjoy our game.

11 Source Code

All of the files for the project are uploaded to Github: https://goo.gl/Tu5oHV