ECE 385

Fall 2019

Final Project

Animated Graphical Game on FPGA: "Fish Hunter"

Xinglong Sun Churan He

ABD T 8AM

Mihir Iyer

Contents

1. Introduction

For our ECE385 Final Project, we designed and built a complex graphics game "Fish Hunter" on the DE2-115 FPGA Development board. Players can use USB keyboard to rotate a cannon located at the bottom of the "ocean" and control it to shoot fishnet to capture moving fishes in our animated ocean background. Concretely, our game includes fish of various types and animations, and they contain different points to be added to the player's current total score. Moreover, every fish is randomly generated and performs random movements, which makes our game environment more vivid and versatile. We also add a page to display the 5 highest scores to give players a sense of how good they perform in the game. Since our game requires lots of mathematical computations, random generations, and collision detections, we choose to implement a large portion of the game logic using NIOS II and C. We built a very powerful graphics display system in SystemVerilog that is able to handle all sorts of low-level cumbersome drawing tasks, so that we can focus more on the game design in C. We did lots of planning to split the work between hardware and software and coordinate them to work together perfectly.

2. Game Explanation

Our game "Fish Hunter" is a 2D keyboard control multiple color video game in which the user can use a keyboard to control the cannon, shoot the cannonball and catch different kinds of fishes.

In the main menu, user can use a keyboard to choose either start the game or view high score records. If the user selects playing the game, the cannon would be displayed and six kinds of fishes would be randomly generated. (Figure 1)

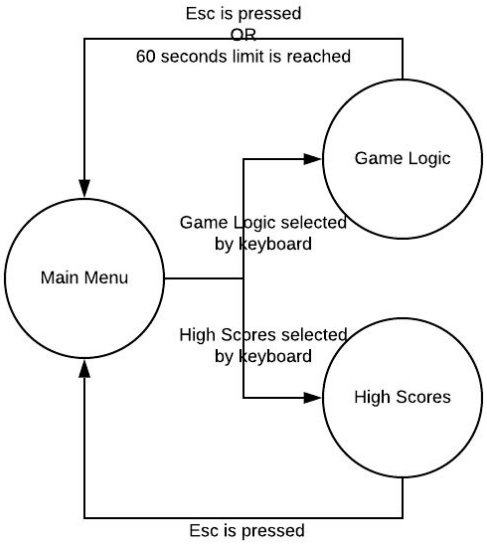


Figure 1 Game state transition diagram

In the game, the user has a 60 seconds time limit, and the score is accumulated according to the points value of specific kind of fish. The user can use “A” or “Left Arrow” key to rotate the cannon to the left, and use “D” or “Right Arrow” key to rotate the cannon to the right. If the user has aimed to the target, he can press down and hold the “Space Bar” to determine the power of the cannon. The more power cannon has, the more distance a cannonball can be shot. There are 8 levels of power, and the cannon ball can cover all places in the pool. The cannonball would be shot to the target area determined by cannon’s direction and power, and then it would explode to be a net. The fishes within the rage of the net would be captured. If a fish is captured, it would disappear with the net and its points would be added to the user’s score. The game would be automatically stop and go back to the main menu when the 60 seconds time limit is reached, and at the same time the high score records would be refreshed. If “Esc” key is pressed, the game would also be stopped and go back to the main menu no matter if the time remains.

If the user selects to view the high scores, 5 top scores will be labeled and displayed from top to bottom. If “Esc” key is pressed, the user could go back to the main menu.

3. Overall Design Procedure and Consideration

3.1. Hardware Design Procedure in SystemVerilog

Since this project is of comparably big size, we try to modularize our design as much as possible. We start to first implement the VGA graphics display. Several possible displaying methods we explored include: Fixed-Function Approach, Single Buffer, and Double Buffer. Fixed-Function uses the least amount of memory but is limited to display simple graphs. Since we wanted our fish to look as vivid as possible, we choose to implement one of the on-chip-buffer approaches with sprite sheet resources preloaded to SRAM. Compared to the single buffer where our screen is very prone to blink at frame transition, double buffer produces more stable and smoother display although it occupies more on-chip memory.

Through computations, we found that with double buffer built on on-chip memory, we can only include 32 colors in our sprite sheet. The detail of the computations is mentioned in the later subsection. To achieve this need without compromising our graphics quality, we used an unsupervised machine learning approach, K Means Clustering, to compress the colors appearing in our sprite sheet. We encoded the hex color to an integer identifier and stores this identifier into the SRAM instead of storing the three RGB values. In this way, we can save the memory usage by 66%. We then perform corresponding decoding in SystemVerilog to extract the RGB values which are later fed to the VGA screen.

At this point, we shift our focus to C programming in NIOS II after we finished building the necessary interface.

3.2. Interface Design Procedure

Our draw engine, VGA controller and resources are in hardware, whereas our game logic is defined in software. In order to establish the connection between hardware and software, we need to build a memory mapped hardware module as an interface, avalon_draw_interface, to share necessary data values.

With the interface, the software can tell the hardware what pattern should be drawn at what specific place on the screen. After drawing, the hardware needs to send a feedback signal to the software representing the drawing process has been finished. If every object of a frame has been drawn, the software will send a frame done signal to hardware, then the hardware can sync the double frame buffer and display the frame on the screen.

3.3. Software Design Procedure in C

To separate our software into different components and hierarchies, we again modularized our code as much as possible. We keep the group of subroutines that serve the same high-level function in the same ".c" file. Since we have lots of structures in our game, such as Fish, Cannon, CannonBall, FishNet, etc., we choose to build a struct for each of them. Besides struct, we also implement a base prototype of each struct to quickly populate objects. We first chose to achieve the moving and animation of fish by implementing an "animal struct" that accepts different kinds of fish as parameters. Details of animation are explained in the next subsection. We then chose to implement the randomness, which contains the random generation and random movement. We simply make use of the random generator in C, which is a linear congruential generator. The limit of this generator is that it will produce the same result at every first use (after booting up FPGA). However, it will later produce varying and randomized results which satisfies our requirements. We then started to build our control and Cannon motion. This part includes a relatively large amount of math computation and analysis, which will be elaborated later. Many miscellaneous things like score keeping, time counting, main menu, and high score table are added later to make our game more complete.

Several Points Worth Noting:

3.3.1. Animations

Because we want to make our game more vivid, we decide to add animation effects to fish, the cannonball, and the fish net. The animation for each object contains 4 - 8 frames, and all of these frames have to be pre-stored in the sprite sheet. If we need to draw a specific animated object on the screen, we would use these frames in a sequence, from the first frame to the last frame. Therefore, the animation effect is achieved by switching each animation frame. If the last frame is reached, the next frame would be a loop back to the first frame. Once the screen needs to be updated, we draw the next frame for each object. Because we use C program to control the draw engine, each object(Struct) has an array of its frames indexes which acts as a circular buffer.

3.3.2. Mathematical Computation

3.3.2.1. Trajectory Computation of Cannonball

As we want our cannonball moving trajectory to follow the inclination of the cannon, we need some basic trigonometric calculations to make this work. In other words, we want the cannonball to be shot straight out of the cannon. As we mentioned above, the rotation of the cannon is achieved by switching frame between 23 images of cannon positioned at different angles. We keep the current inclination angle of the cannon in the struct as an attribute when we compute trajectory. We specify the total velocity of the cannonball as 30, which is determined by comparing with the fish moving velocity. Suppose our current inclination angle is θ, we simply set our horizontal velocity $x_{velo} = 30 * cos(\theta)$ and set our vertical velocity $y_{velo} = -1 * 30 * sin(\theta)$ (Scaling by -1 because our cannonball moves up from the bottom(y = 480)). At each frame after the cannonball has been shot, we increment the x_position of the cannonball by x_velo amount and y_position of the cannonball by y_velo amount. In this way, the slope of the path of cannonball will be the same as $tan(\theta)$.

3.3.2.2. Max Possible Travelling Distance of Cannonball

Because the 8 levels power value determines distance of the cannonball, we need to find the max possible travelling distance at the specific cannon's angle. To analyze this problem, we need to discuss separately for the cannon's direction in Region A and Region B (specified in Figure 2).

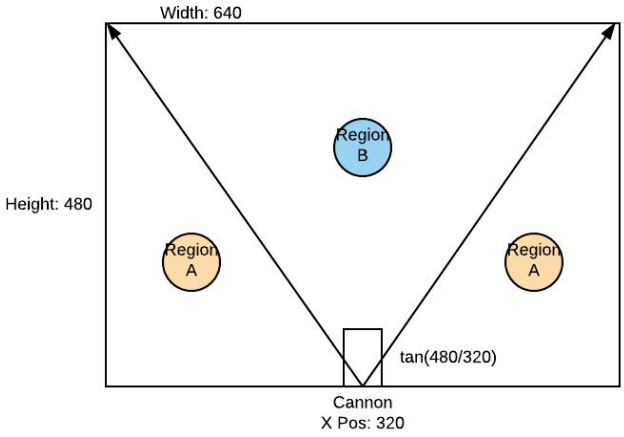


Figure 2 Cannon region specification

If the cannon's direction is in Region A, we would use 320 (half of the width), and the angle of cannon to determine the max travelling distance using the equation:

$$d = \frac{320}{cos(angle)} \left(angle \subset \left[0, tan^{-1}\left(\frac{480}{320}\right) \right] \bigcup \left[180 - tan^{-1}\left(\frac{480}{320}\right), 180 \right] \right)$$

If the cannon's direction is in Region B, we would use 480 (height), and the angle of cannon to determine the max travelling distance using the equation:

$$d = \frac{480}{sin(angle)} \left(angle \subset \left(tan^{-1}\left(\frac{480}{320}\right), 180 - tan^{-1}\left(\frac{480}{320}\right) \right) \right)$$

Once we get the max traveling distance of the cannonball, the actual traveling distance can be easily calculated by multiplying the max traveling distance by power/8. Once the cannonball reaches the distance, it would stop and then become the fishnet.

3.3.2.3. Overlapping (Catching) Detection

Once the fishnet is deployed, we need to detect if there is any fish captured by the fishnet. The criteria of being captured is that the midpoint position of the fish is in the region of a deployed fishnet.

In the actual calculation, we need to calculate the difference between the center point of the fish the the center point of the fishnet on X axis and Y axis, delta X and delta Y (as shown in Figure 3).

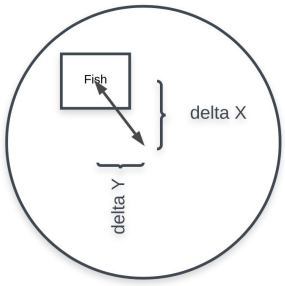


Figure 3 Fish Capturing Detection

Then, we need to check if $(delta\ X)^2 + (delta\ Y)^2$ is smaller than the $(radius\ of\ fishnet)^2$, in this case 2139. If the fish is captured, the fish would stop moving and start blinking, until vanishes along with the fishnet. Captured fish would be reborn at the edge of the display, so there will always be enough fishes for the user to capture. The points of captured fish would be added to the user's total score.

3.3.3. Randomness

Randomized design is the key attribute in every attractive game, and it's also an important element in our game. Two aspects of our game that are randomized include the generation of fish and the movement of fish. More particularly, for the generation of fish, numbers of different types of fish, birth-positions and rebirth-positions, and rebirth-types are all randomized. For the movement of fish, moving velocity and moving trajectory of each fish are randomized. In other words, at the beginning of our game, there will be a random number of fish of different types showing up in the environment. However, the random number is adjusted subtly to make sure that fish of lower points appear more frequently than more valuable fish. Moreover, the velocity of each fish is randomized within a constrained range. Therefore, we can make sure that the highest possible velocity of the cheap fish is still lower than the more valuable fish, which is supposed to be harder to catch. After a fish is caught, a fish of random type will be

regenerated according to the rarity-frequency distribution of each fish. Moreover, each fish is given a random trajectory function that determines how it moves in the ocean. Several possible trajectories include a straight line, a sloped line, and sine/cosine waves. With these randomized elements, our game becomes more versatile and playable.

3.3.4. Score & Time

At the same frame when we evaluate whether the fish is caught, we increment the total score by the point allocated to that specific fish type. For the time counting, we used the Altera Core TIMER in our platform designer and deploys its functions in our C program to count how many seconds have passed since the beginning of the game. We include necessary fonts in our sprite sheet to visually display the current score and time.

3.4. Graphical Design

3.4.1. Sprite Sheet Resources & Generation

In order to display complex patterns on the screen, we need to put every object to the sprite sheet. The sprite sheet we used is a large graph (as shown in Figure 4) of 640 * 1625 pixels. It contains the background, cannon of different directions, cannonball, fishnet, various fishes, and fonts. The sprite sheet is stored in SRAM in a compacted form (using 5 bits binary to represent 32 colors), and we use Blitter module to access the specific pattern in the spite sheet.

The Blitter module can only fetch rectangle pattern from the sprite sheet, however most of the objects in our game have irregular shape. So we determine to use a specific color (Red: 0xfe0000) as the background of the sprite sheet. This color will be filtered out when writing to the frame buffer, thus we can draw every pattern on the screen without any shape limitation.

Figure 4 Sprite Sheet of "Fish Hunter"

3.4.2. Python Code Description

We wrote some Python code to assist us to compress the number of colors appearing in our sprite sheet, generate RAM/HEX later loaded to the memory, and generate the hardware decoder in System Verilog Syntax. In this section, we will briefly discuss how we achieve these functions.

Like we mentioned above, because of the limitations of on-chip memory, we can only include 32 colors in our sprite sheet. This requires us to figure out an approach to reduce the unique colors in our image without compromising the image quality.

```python
def kMeans_quantizer(image,k):
    im_shape = image.shape
    n_rows = im_shape[0]
    n_cols = im_shape[1]
    #create k-means object
    kmeans = KMeans(n_clusters = k)
    #reshape pixel value to be like data points
    if len(im_shape) == 2:
        pixel_vals = np.array([[image[row,col]] for row in range(n_rows) for col in range(n_cols)])
    else:
        pixel_vals = np.array([image[row,col] for row in range(n_rows) for col in range(n_cols)])
    color_labels = kmeans.fit_predict(pixel_vals)
    q_image = np.zeros(im_shape).astype(np.uint8)
    colors = kmeans.cluster_centers_.astype(np.uint8)
    for i,label in enumerate(color_labels):
        q_image[int(i/n_cols),i % n_cols] = colors[label]
    plt.imshow(q_image)
    return q_image, colors
```

Figure 5 K-Means Quantizer

Picture attached above (Figure 5) shows the KMeans_quantizer we implemented. We used KMeans from the scikit-learn package. This function will pick 32 centroid-colors that minimizes the MSE of the transformed image(constructed using the 32 colors) and the original image.

After we picked the 32 colors, we encoded the color (Figure 6) of each pixel in the image simply using the index of this color in the array. We also wrote a function to generate required decoder code in SystemVerilog since writing 32 groups of SV codes by hand will be tiresome.

```python
def genSV(hex_color,filename):
    print(hex_color)
    sv = open(filename+"SV.txt","w")
    counter = 0
    for color in hex_color:
        """
        print("16'd"+str(counter)+":")
        print("begin")
        print("    Red = 8'h"+str(color[0][2:])+";")
        print("    Green = 8'h"+str(color[1][2:])+";")
        print("    Blue = 8'h"+str(color[2][2:])+";")
        print("end")
        print('\n')
        """
        sv.write("16'd"+str(counter)+":\n")
        sv.write("begin\n")
        sv.write("    Red = 8'h"+str(color[0][2:])+";\n")
        sv.write("    Green = 8'h"+str(color[1][2:])+";\n")
        sv.write("    Blue = 8'h"+str(color[2][2:])+";\n")
        sv.write("end\n")
        sv.write('\n')
        counter += 1
    sv.close()
```

Figure 6 SV Decoder Generator

Our third handy helper function is the RAM/HEX generator. This is achieved by simply writing each “color index”(encoded identifier) in hex sequentially in the “.ram” file.

4. Hardware Explanation

4.1. Written Description

Our hardware part contains a complete graphical drawing system and the USB Keyboard interface. In particular, this can be further broken down into a graphical drawing engine and graphical display related components. Our graphical drawing engine works at the on-board frequency, 50MHz. It accepts queries sent from the NIOS II processor and completes the specified low-level tasks. This contains Blitter related components and the Frame Buffer Controller. Our graphical display related components handle all the VGA monitor related stuff, such as generating VGA clock, matching RGB colors fed to the VGA screen, etc. For the USB related part, we continue to use the hardware components from Lab 8, which mainly consists of a hpi_to_intf interface and leaves the specific stuff for the software to accomplish.

4.2. Block Diagram

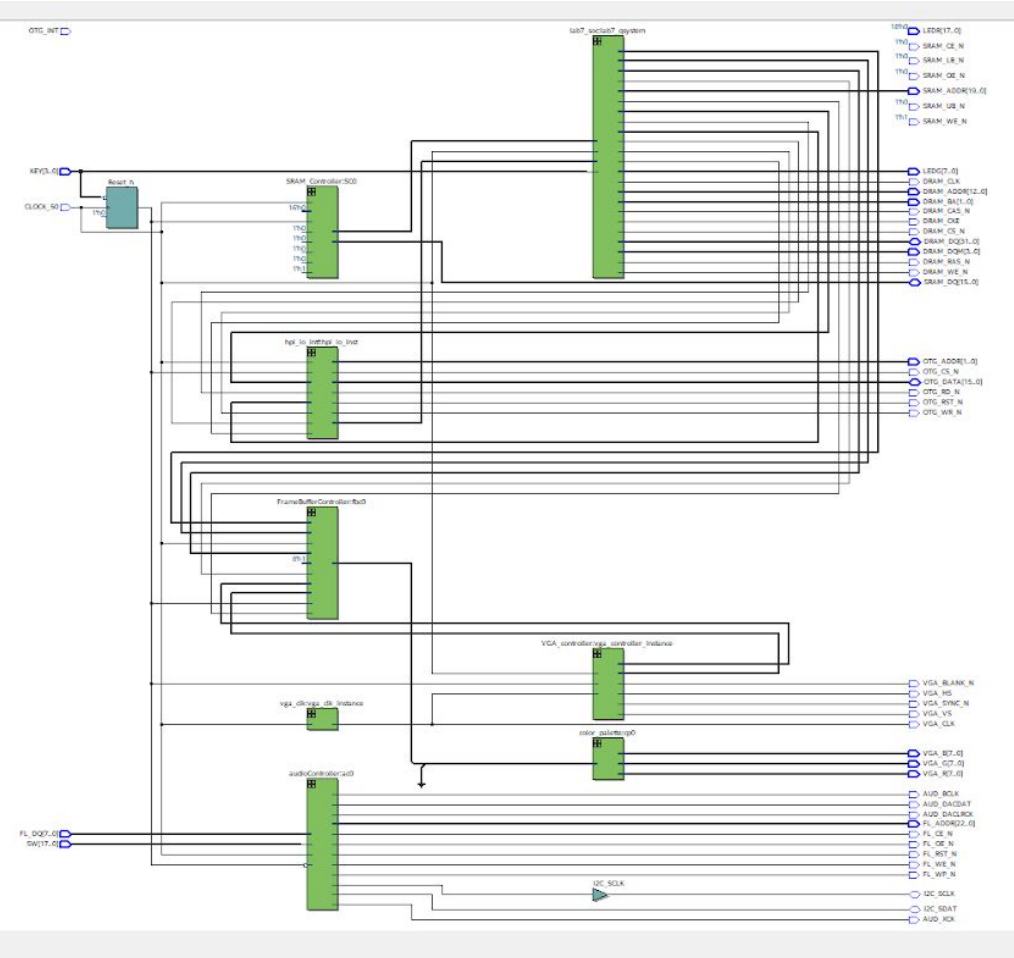


Figure 7 Top-level Netlist Viewer

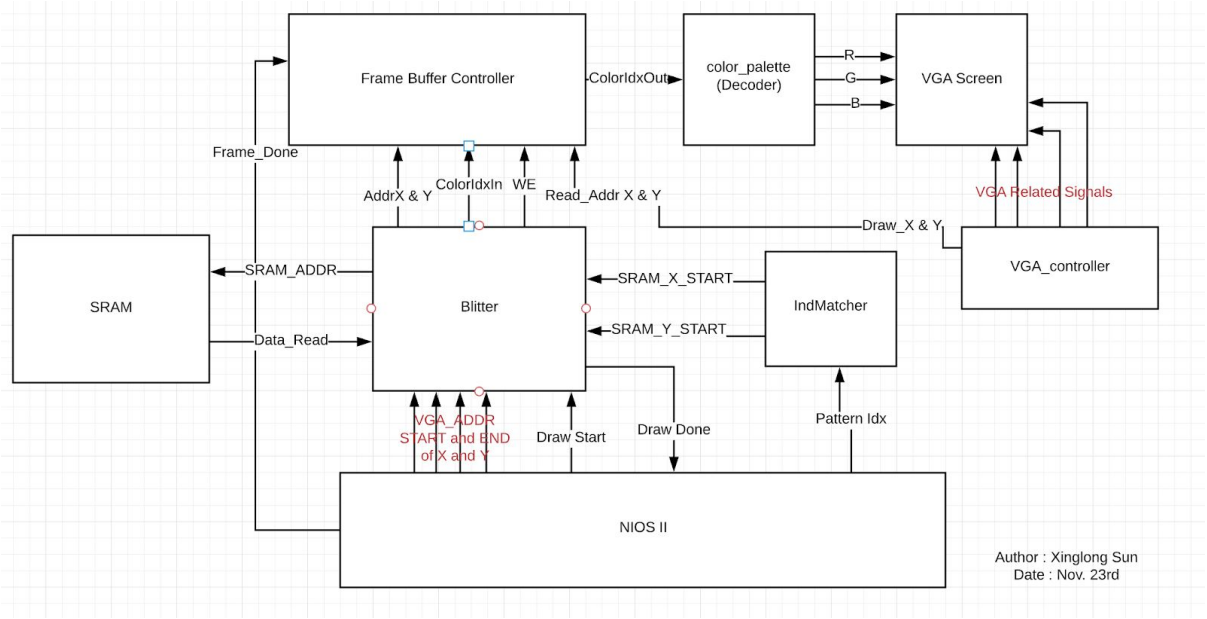


Figure 8 Data Flow Diagram

4.3. Hardware Module Description

4.3.1. Top-level

FishHunter.sv

Inputs:

logic CLOCK_50,
logic [3:0] KEY,
logic [17:0] SW,
//For USB Interface:
logic OTG_INT

Inouts:

//For USB Interface:
inout wire [15:0] OTG_DATA,
//For SRAM Interface:
inout wire [15:0] SRAM_DQ,
//For SDRAM Interface:
inout wire [31:0] DRAM_DQ,
//For Flash Interface:
inout wire [7:0] FL_DQ,

Outputs:

logic [7:0] LEDG,
//For VGA Interface:
logic [7:0] VGA_R, VGA_G, VGA_B,
logic VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS,

//For USB Interface:
logic [1:0] OTG_ADDR,
logic OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N,
//For SRAM Interface:
logic SRAM_CE_N, SRAM_UB_N, SRAM_LB_N, SRAM_OE_N, SRAM_WE_N,
logic [19:0] SRAM_ADDR,
//For SDRAM interface:
logic [12:0] DRAM_ADDR,
logic [1:0] DRAM_BA,
logic DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK
logic [3:0] DRAM_DQM
//For Flash interface:
logic FL_CE_N, FL_OE_N, FL_WE_N, FL_RST_N, FL_WP_N,
logic [22:0] FL_ADDR

Purpose & Description:

Same as previous projects, we need to implement the top level of hardware components, so all parts could have appropriate connections. The top level of our hardware contains NIOS II SOC, Audio Controller, VGA clock, VGA controller, Frame Buffer and Color Palette, and connects these hardwares with other off-chip components including Clock, switches, keys, USB, VGA, SRAM, SDRAM and Flash. The RTL diagram is attached above as Figure 7.

4.3.2. Blitter

As mentioned above, blitter acts as a drawing core, accepting queries sent from software and outputting corresponding graphs to the VGA screen. Concretely, blitter will copy values from a specific memory region into the frame buffer, which will later be displayed on the screen. Blitter module is essentially a FSM with counters. Two counter modules we created to assist our design are VGA counter, which mimics the beam swiping action and loops through the region of the frame buffer specified by the software, and SRAM counter, which loops through the sprite sheet stored in SRAM and extract the specified object pattern. Explanation of each single module will be provided below:

SRAM_Counter.sv

Inputs:

logic ctr_reset_sram, Clk, INIT_SRAM, hold_sram
logic [10:0] SRAM_ADDR_X_Start, SRAM_ADDR_X_Stop,SRAM_ADDR_Y_Start, SRAM_ADDR_Y_Stop

Outputs:

logic [19:0] SRAM_ADDR

Purpose & Description:

This module implements a counter with initial and stop values. It's a helper module that facilitates the computation of SRAM address to access at each clock cycle after a draw query is sent from the software. When ctr_reset_sram is set to high, the counter value will be cleared. When INIT_SRAM is set high, X_count and Y_count (two internal values that are used to calculate SRAM_ADDR) are set to SRAM_ADDR_X_Start and SRAM_ADDR_Y_Start. When hold_sram is set to high, the counter will remain its current value. At each clock cycle, X_count will keep incrementing until it hits SRAM_ADDR_X_Stop. At this point, it will be reset back to zero, and Y_count will be incremented by 1. The above process keeps repeating until Y_count reaches SRAM_ADDR_Y_Stop.

With X_count and Y_count defined as above, we have SRAM_ADDR calculated as follows:

$$SRAM\_ADDR = Y_{count} * 640 + X_{count}$$

640 is the width of our sprite sheet and should be changed if a sprite sheet of new width is used.

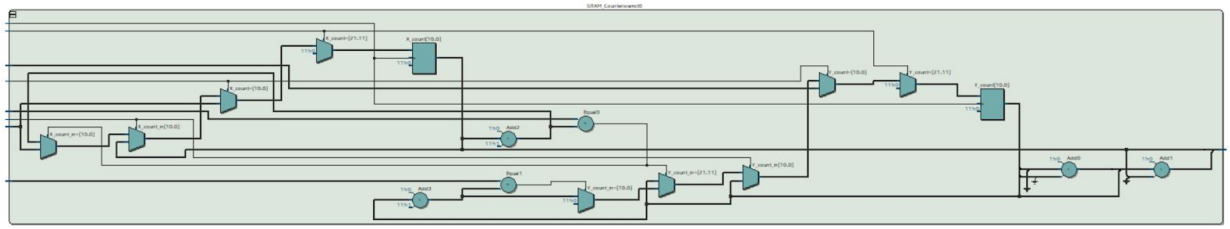


Figure 9 RTL Diagram for SRAM Counter

VGA_Counter.sv

Inputs:

logic ctr_reset_vga, Clk, INIT_VGA, hold_vga,
logic [10:0] AddrX_Start, AddrY_Start, AddrX_Stop, AddrY_Stop

Outputs:

logic [10:0] AddrX, AddrY

Purpose & Description:

This module implements a counter with initial and stop values. It's a helper module that facilitates the computation of frame buffer address to write to at each clock cycle after a draw query is sent from the software. This module is basically the same as the SRAM_Counter explained above but produces both X_count and Y_count as its outputs(AddrX and AddrY), which are later fed to the frame buffer as the write address.

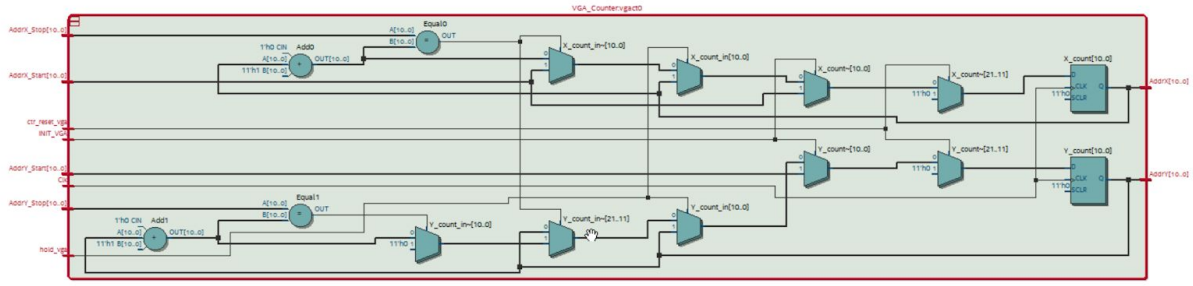


Figure 10 RTL Diagram for VGA Counter

Blitter.sv

Inputs:

logic Clk, Reset_h, Draw_Start,
logic [15:0] Data_Read, //Data read from SRAM
logic [10:0] AddrX_Start, AddrY_Start, AddrX_Stop, AddrY_Stop,
SRAM_ADDR_X_Start, SRAM_ADDR_Y_Start,
logic [7:0] Color_Filter,

Outputs:

logic [10:0] AddrX, AddrY, //Location on the buffer
logic [19:0] SRAM_ADDR, //Location on the SRAM
logic [7:0] ColorIdxIn, ////Data input to the Frame Buffer
logic We, Draw_Done

Purpose & Description:

The blitter module contains the two helper counter modules described above and implements a FSM that provides control signals for these two counters. The FSM includes five states, Reset, INIT, Read_From_SRAM, WRITE_TO_VGA, DONE. In the Reset state, VGA and SRAM counter values are reset to zero. In the INIT state, VGA and SRAM counters are initialized to the start values coming from the input ports by raising INIT_VGA and INIT_SRAM high. The next state, Read_From_SRAM, will be the first operation state and will set hold_vga to high. In WRITE_TO_VGA, hold_sram and We are set to high. Read_From_SRAM is kind of like a dummy state, which leaves the necessary clock delay time for SRAM, which is 10ns on DE2_115 board. With 10ns clock delay, the data read out from the SRAM is guaranteed to be stable in the next clock period. We is raised high during the WRITE_TO_VGA to enable the data to be written into the frame buffer. The above two states will repeat until AddrX and AddrY hit AddrX_Stop - 1 and AddrY_Stop - 1. The final state in a full working cycle is DONE, where Draw_Done is set high to inform the software of the completeness of the drawing cycle. After Draw_Start has been reset back to 0, the state machine loops back to the first state. One thing to notice is that AddrX_Stop and AddrY_Stop are both one pixel past the actual stop locations because in this way we can simply pass in start + width/height as the stop locations from software.

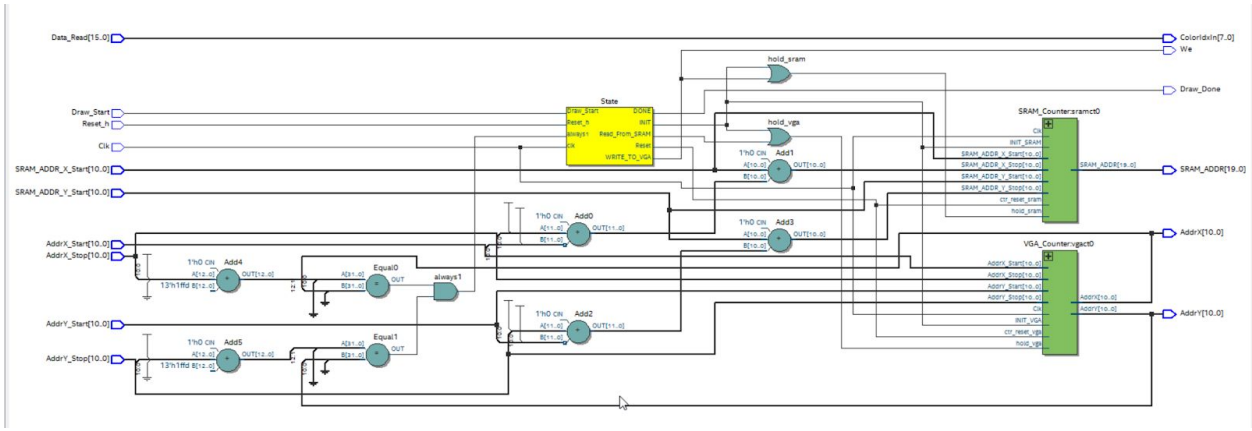
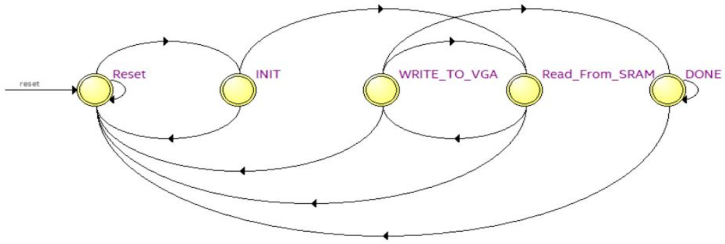


Figure 11 RTL Diagram for Blitter



	Source State	Destination State	Condition
1	DONE	DONE	(Draw_Start).(!Reset_h)
2	DONE	Reset	(!Draw_Start) + (Draw_Start).(Reset_h)
3	INIT	Read_From_SRAM	(!Reset_h)
4	INIT	Reset	(Reset_h)
5	Read_From_SRAM	Reset	(Reset_h)
6	Read_From_SRAM	WRITE_TO_VGA	(!Reset_h)
7	Reset	Reset	(!Draw_Start) + (Draw_Start).(Reset_h)
8	Reset	INIT	(Draw_Start).(!Reset_h)
9	WRITE_TO_VGA	DONE	(always1).(!Reset_h)
10	WRITE_TO_VGA	Read_From_SRAM	(!always1).(!Reset_h)
11	WRITE_TO_VGA	Reset	(Reset_h)

Figure 12 State Machine Diagram for Blitter

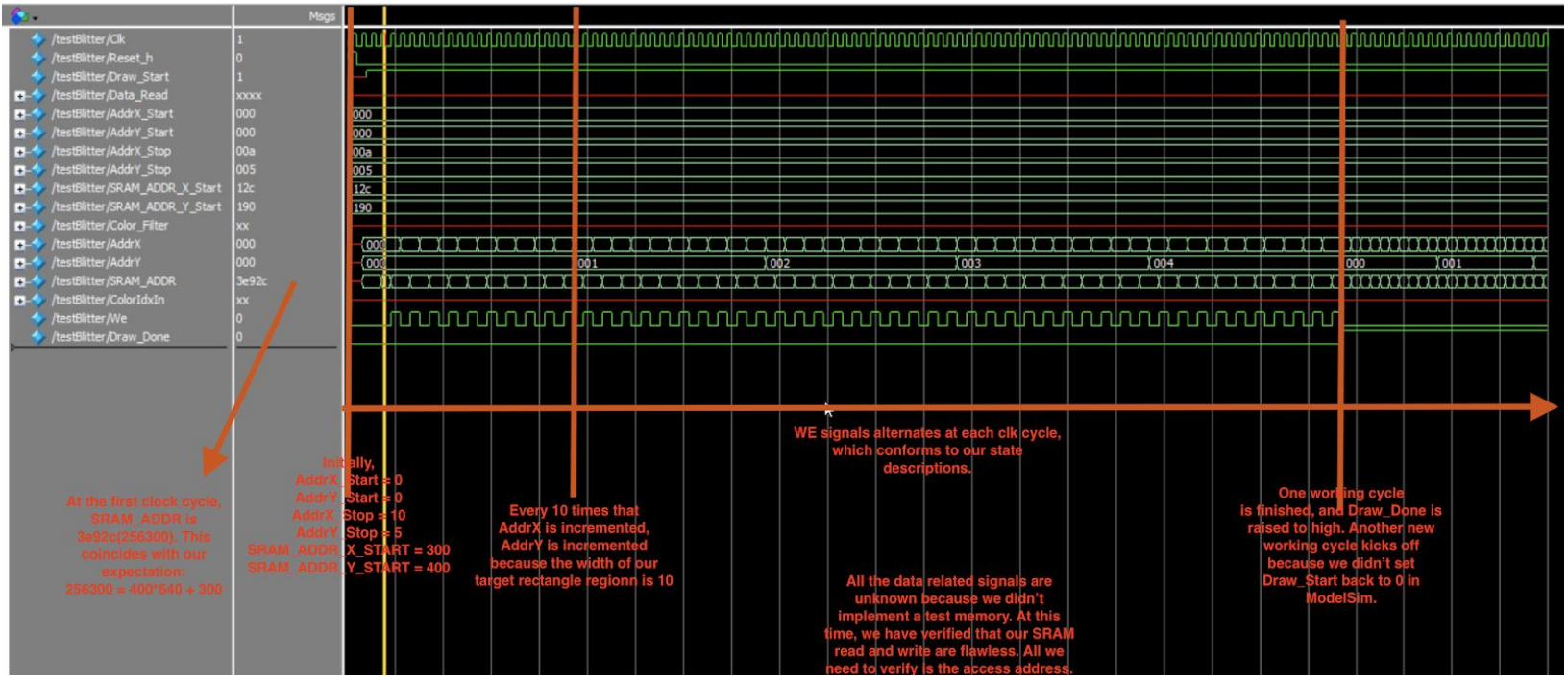


Figure 13 Annotated Waveform for Blitter Simulation

4.3.3. Frame Buffer Controller (Double Buffer)

As we mentioned above, we implement a double buffer approach to achieve a more stable and smoother visual display. To control and coordinate our two frame buffers, we wrote a controller which is essentially a FSM to provide essential control signals to the buffers.

FrameBuffer.sv

Inputs:

logic Clk, We, Reset_h,
logic [10:0] AddrX, AddrY, Read_AddrX, Read_AddrY,
logic [4:0] ColorIdxIn, Color_Filter

Outputs:

logic [4:0] ColorIdxOut

Purpose & Description:

This is a fundamental module that declares a portion of the on-chip memory and uses it as the frame buffer. Since we read and write at the same time, we keep the read and write address ports(AddrX, AddrY, Read_AddrX, Read_AddrY) and read and write data ports(ColorIdxIn, ColorIdxOut) separate. We explained above how we use a certain color filter to make the background transparent and draw irregular shapes. At every clock cycle, ColorIdxOut is always equal to the data accessed by the Read_AddrX and Read_AddrY. Only when We is high, location indicated by AddrX and AddrY is filled with ColorIdxIn. The on-chip memory size is 5 * 640 * 480 for each frame buffer. Data size 5 corresponds to the color index length, and 640 *

480 is the same as the VGA monitor size. We do an analysis beforehand to determine the data length to use the on-chip memory at its max limit.

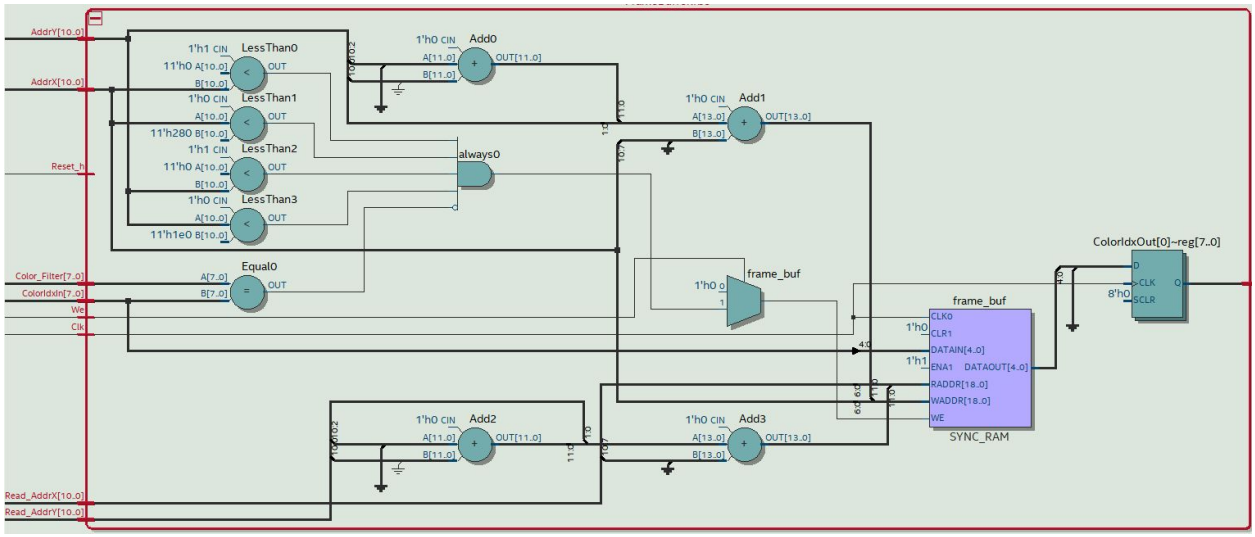


Figure 14 Frame Buffer RTL Diagram

FrameBufferController.sv

Inputs:

logic Clk, We, Reset_h, Frame_Done,
logic [10:0] AddrX, AddrY, Read_AddrX, Read_AddrY,
logic [7:0] ColorIdxIn, Color_Filter

Outputs:

logic [7:0] ColorIdxOut

Purpose & Description:

This is the top level for our frame buffer controller module. It wraps up two frame buffers and make the appropriate connections between them. Our first frame buffer, fb0, is the connecting bridge between fb1 and blitter. The second frame buffer, fb1, is used as the visual display. We include a counter same to our VGA counter module described above and use it to swipe and copy data from fb0 to fb1. Therefore, the top-level write addresses, AddrX and AddrY, are connected to the cache frame buffer fb0. We feed the counter values, X_count and Y_count, as the write addresses into the actual display buffer fb1. The signal Frame_Done indicates the frame transition. Concretely, after we send draw queries of every object in the same frame from software, we set Frame_Done to high and set it back to low when a new frame begins. In this way, the write enable signal of fb0 is directly connected to the top-level We, and the write enable signal of fb1 is Frame_Done. For read addresses, the top-level Read_AddrX and Read_AddrY feed into ports on fb1, and read address of fb0 will be the same as the counter values, X_count and Y_count. A detailed block diagram showing the connections is provided below.

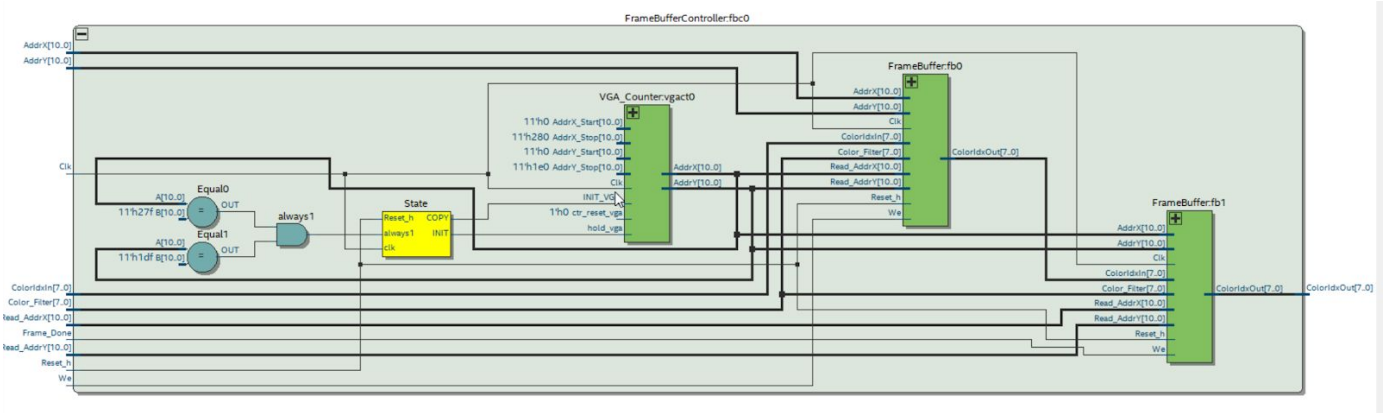


Figure 15 Frame Buffer Controller RTL Diagram

4.3.4. Index Matcher

IndMatcher.sv

Inputs:

logic [9:0] Draw_Idx,

Outputs:

logic [10:0] SRAM_ADDR_X_Start, SRAM_ADDR_Y_Start

Purpose & Description:

When the software want to draw a specific object on the display, it would send the index of that object to the avalon_draw_interface. Then the Index Matcher can provide the address of the object in SRAM to the Blitter in order to load the correct pattern.

The functionality of the Index Matcher is simple. It takes the index of the object, and output the SRAM_ADDR_X_Start and SRAM_ADDR_Y_Start which are pre-defined in the .sv file.

4.3.5. Graphical Display Modules

Along with the blitter module, if we want to display patterns on the VGA screen, we also need a VGA Controller which provides standard VGA protocol and the color palette which convert the compressed color index to its original RGB value.

VGA_controller.sv

Inputs:

logic Clk, Reset, VGA_CLK,

Outputs:

logic VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N,

logic [10:0] DrawX, DrawY

Purpose & Description:

VGA_controller is the hardware for the VGA signal protocol. It needs to provide signals for HS and VS, as well as BLANK (implemented with counters). These signals can let a VGA

monitor know that our system is generating a valid VGA signal, thus it can output the display content correctly. The DrawX and DrawY would be connected to the second frame buffer, so we can get the color information at a specific X, Y position.

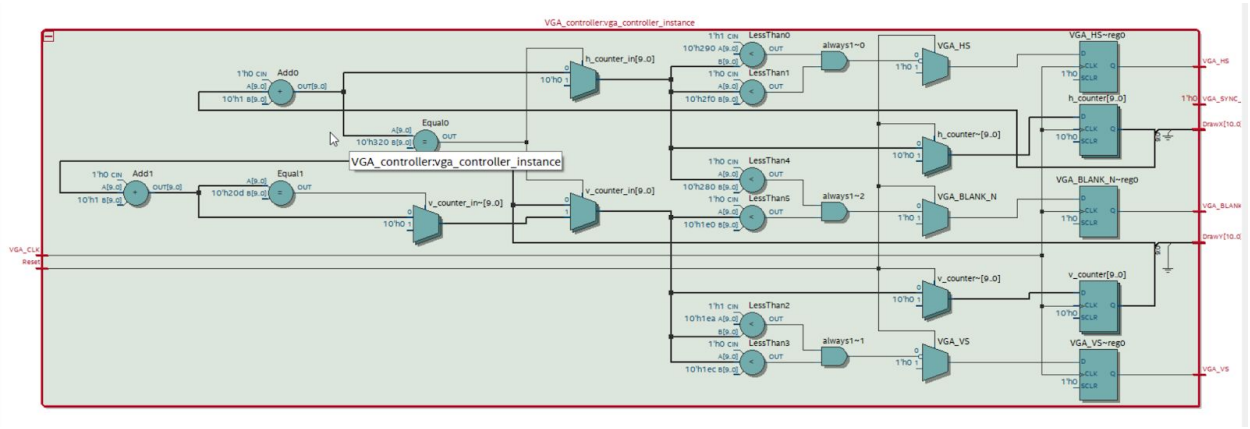


Figure 16 VGA Controller RTL Diagram

color_palette.sv

Input:

logic [15:0] ColorIdx,

Output:

[7:0] VGA_R, VGA_G, VGA_B

Purpose & Description:

The acts as a decoder, extracting the unique RGB colors represented by the ColorIdx. As mentioned above, code for this module is generated in Python along with our sprite sheet RAM. As the VGA controller provide DrawX and DrawY signal to the second frame buffer, the frame buffer would provide the color index of that specific pixel to the color palette module, then the color palette could assign values of R, G, B channels.

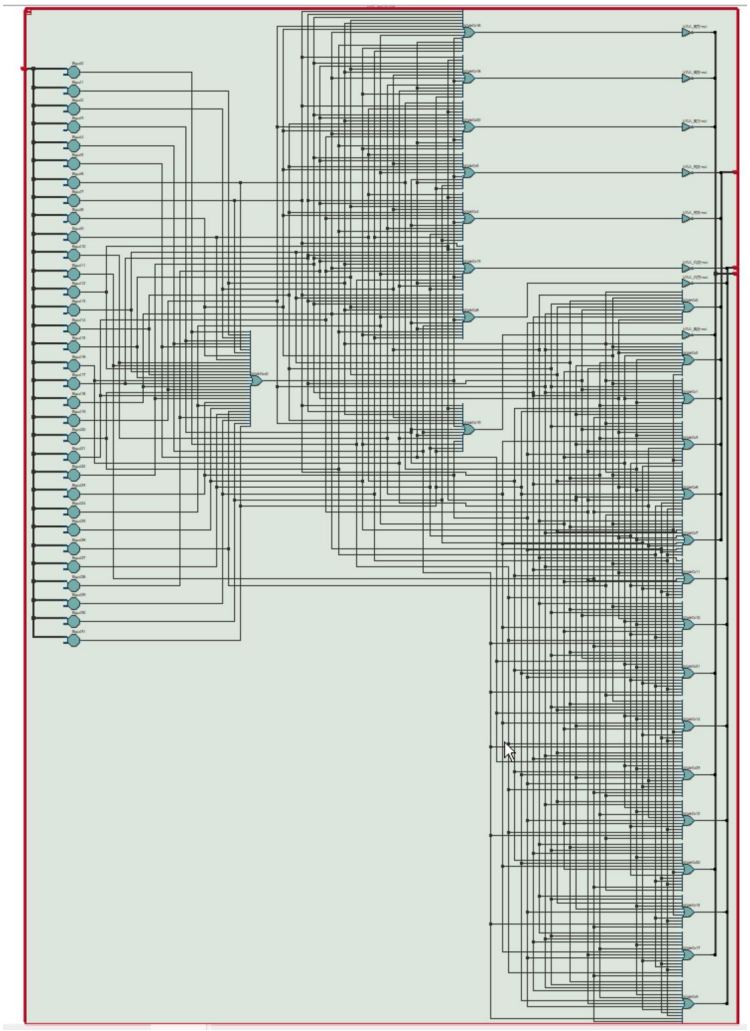


Figure 17 Color Palette RTL Diagram

4.3.6. Audio Related Modules

For the audio part, we learn from an existing audio project by a group of students in Linköping University. However, they store their audio in the on-chip memory, but we have used all of our on-chip memory for the double frame buffer. Therefore, we modify their work and add the Flash controller where we decide to store our music file, which is 16-bit PCM with 8 KHz sampling rate. Because the flash has about 90 ns delay of the output, we need another clock prior to the clock design for the on-chip memory and a buffer to temporarily store the 16-bit audio data. The audio module is able to play our 25 seconds background music within a loop.

audioController.sv

Inputs:

logic clk,reset,SW0,
logic [7:0] FL_DQ,

Inouts:

inout wire SDIN,

Outputs:

logic FL_CE_N, FL_OE_N, FL_WE_N, FL_RST_N, FL_WP_N,
logic SCLK,USB_clk,BCLK,
logic DAC_LR_CLK, DAC_DATA
logic [22:0] FL_ADDR

Purpose & Description:

This part is modified based on the project from the group of students in Linköping University. The original code is written in Verilog HDL, and it uses the on-chip memory for music storage where we don't have enough room. Thus, we replace the on-chip memory module with a flash driver. Because it would take 5 clock cycles for the flash to respond to an address input, and the data width in the flash is only 8 bits. We create a new data clock signal prior to the original original signal for the flash driver, manipulate the address to fetch the first half 8 bits and fetch the second half 8 bits in order, and use a buffer to temporarily store the 16-bit value. Then this 16-bit value would be provided with the original clock signal.

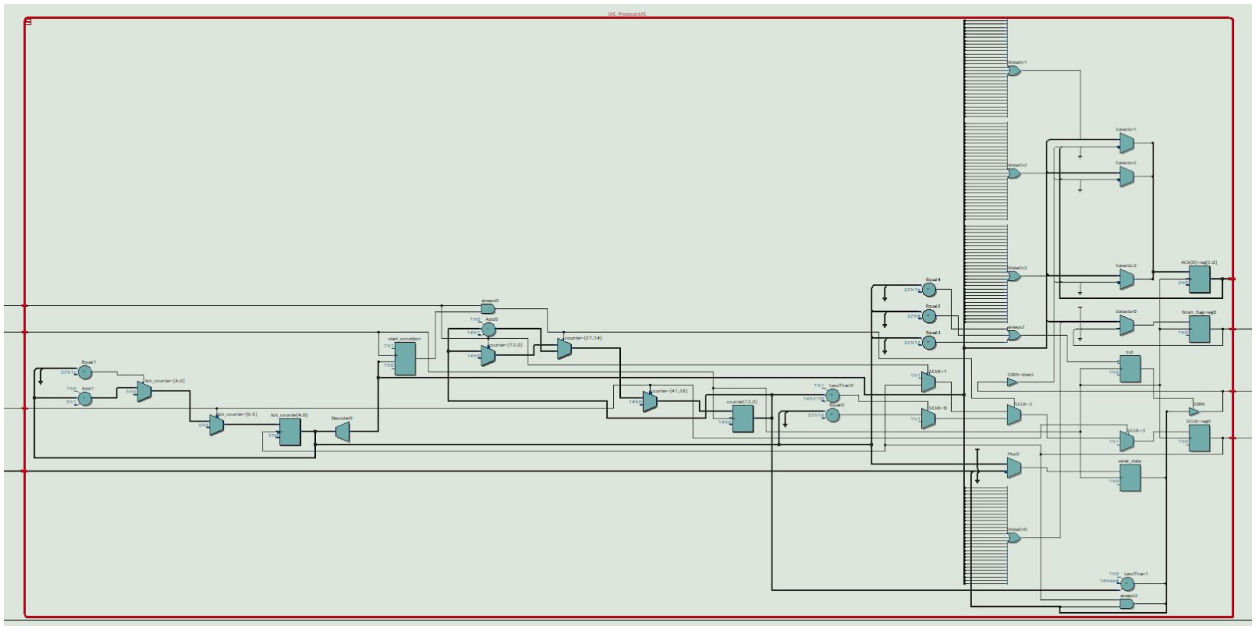


Figure 18 Audio Controller RTL Diagram

I2C_Protocol.sv

Inputs:

logic clk, reset, ignition,
logic [15:0] MUX_input,

Inouts:

logic wire SDIN,

Outputs:

logic finish_flag, SCLK,
logic [2:0] ACK

Purpose & Description:

This part is credited to the group of students in Linköping University. It provides the I2C protocol to let the FPGA to communicate with the off-chip audio DAC chip WM8731.

4.3.7. USB Related Module

hpi_io_intf.sv

Inputs:

Clk, Reset,
[1:0] from_sw_address,
[15:0] from_sw_data_out,
from_sw_r, from_sw_w, from_sw_cs, from_sw_reset

Inout:

[15:0] OTG_DATA

Outputs:

[15:0] from_sw_data_in,
[1:0] OTG_ADDR,
OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N

Purpose & Description:

This serves as an interface between NIOS II and EZ-OTG chip. The details of its descriptions and purposes are explored in the previous sections. Specifically, all the "from_sw_address" labeled inputs are the outputs from our NIOS II software. All the OTG labeled outputs go into the actual pins on the EZ-OTG chip. We have an inout pin OTG_DATA because we both need to read from and write to our USB device registers. Since inout bus should be driven by some register instead of combinational logic, a buffer register is necessary here. Concretely, when "from_sw_w" is HIGH, we assign OTG_DATA to be the same as "from_sw_data_out_buffer". Otherwise, we assign it to HighZ to properly read the value.

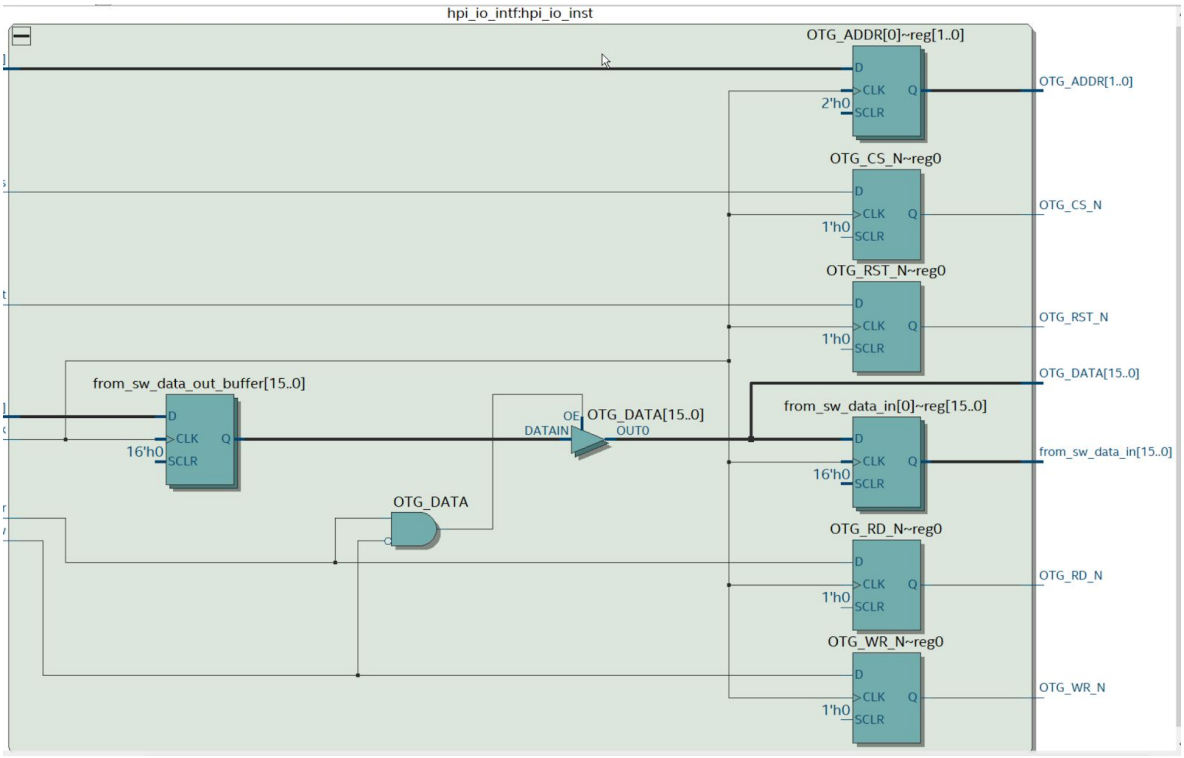


Figure 19 HPI IO Interface RTL Diagram

4.3.8. Hardware/Software Interface

avalon_draw_interface.sv

Inputs:

logic CLK, RESET, AVL_READ, AVL_WRITE, AVL_CS,
logic [2:0] AVL_ADDR,
logic [31:0] AVL_WRITEDATA,
logic [15:0] Data_Read,

Outputs:

We, Frame_Done,
logic [31:0] AVL_READDATA,
logic [10:0] AddrX, AddrY,
logic [19:0] SRAM_ADDR,
logic [7:0] ColorIdxIn

Purpose & Description:

The interface has 8 logical components (as shown in Table 1), which can be accessed by both hardware and software.

Address	Content
DRAW_PTR[0]	Object Index
DRAW_PTR[1]	Draw Start Point on X axis

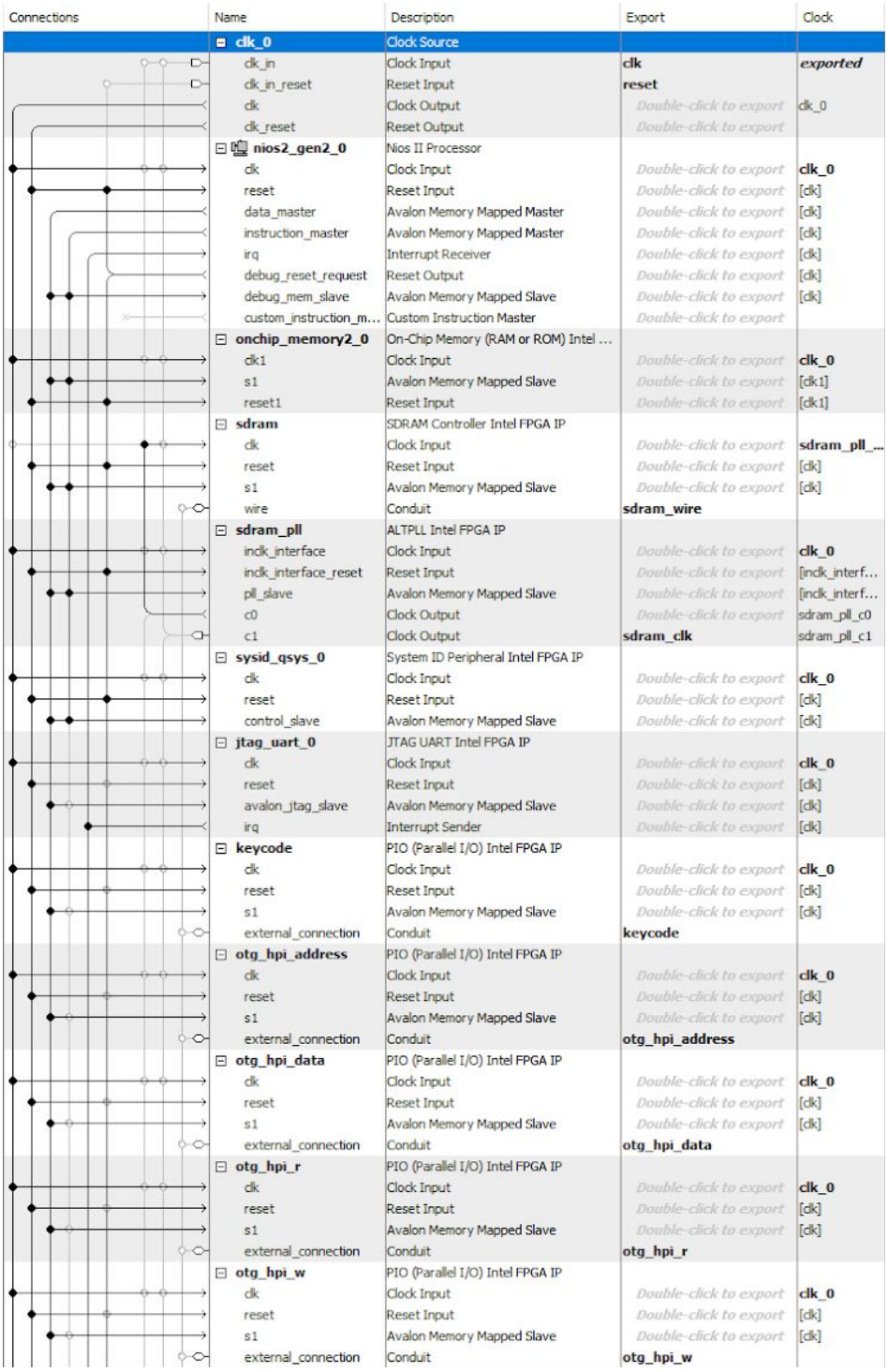
DRAW_PTR[2]	Draw Start Point on Y axis
DRAW_PTR[3]	Draw End Point on X axis
DRAW_PTR[4]	Draw End Point on Y axis
DRAW_PTR[5]	Frame Done
DRAW_PTR[6]	Draw Start
DRAW_PTR[7]	Draw Done

Table 1 Logics in the Interface

When we need to draw any object to the display, the software need to write the Object Index, Draw Start Point on X and Y axis, Draw End Point on X and Y axis, and Draw Start to the interface. When the hardware notices Draw Start become "1", it would read other data and use the Blitter module to draw the object. After drawing the object, the interface would write "1" to Draw Done, so that the software is able to draw the next object. Once all objects on the display are drawn, the software would write "1" to Frame Done, which would sync the content in our double frame buffer. So the display can show every complete frame with tens of different objects.

4.4. NIOS II SOC Description

4.4.1. Qsys Top Level



Connections	Name	Description	Export	Clock
	clk_0	Clock Source		
	clk_in	Clock Input	**clk**	***exported***
	clk_in_reset	Reset Input	**reset**	
	clk	Clock Output	*Double-click to export*	clk_0
	clk_reset	Reset Output	*Double-click to export*	
	nios2_gen2_0	Nios II Processor		
	clk	Clock Input	*Double-click to export*	**clk_0**
	reset	Reset Input	*Double-click to export*	[clk]
	data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]
	instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]
	irq	Interrupt Receiver	*Double-click to export*	[clk]
	debug_reset_request	Reset Output	*Double-click to export*	[clk]
	debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
	custom_instruction_m...	Custom Instruction Master	*Double-click to export*	
	onchip_memory2_0	On-Chip Memory (RAM or ROM) Intel ...		
	clk1	Clock Input	*Double-click to export*	**clk_0**
	s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]
	reset1	Reset Input	*Double-click to export*	[clk1]
	sdram	SDRAM Controller Intel FPGA IP		
	clk	Clock Input	*Double-click to export*	**sdram_pll_...**
	reset	Reset Input	*Double-click to export*	[clk]
	s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
	wire	Conduit	**sdram_wire**	
	sdram_pll	ALTPLL Intel FPGA IP		
	inclk_interface	Clock Input	*Double-click to export*	**clk_0**
	inclk_interface_reset	Reset Input	*Double-click to export*	[inclk_interf...
	pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_interf...
	c0	Clock Output	*Double-click to export*	sdram_pll_c0
	c1	Clock Output	**sdram_clk**	sdram_pll_c1
	sysid_qsys_0	System ID Peripheral Intel FPGA IP		
	clk	Clock Input	*Double-click to export*	**clk_0**
	reset	Reset Input	*Double-click to export*	[clk]
	control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
	jtag_uart_0	JTAG UART Intel FPGA IP		
	clk	Clock Input	*Double-click to export*	**clk_0**
	reset	Reset Input	*Double-click to export*	[clk]
	avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
	irq	Interrupt Sender	*Double-click to export*	[clk]
	keycode	PIO (Parallel I/O) Intel FPGA IP		
	clk	Clock Input	*Double-click to export*	**clk_0**
	reset	Reset Input	*Double-click to export*	[clk]
	s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
	external_connection	Conduit	**keycode**	
	otg_hpi_address	PIO (Parallel I/O) Intel FPGA IP		
	clk	Clock Input	*Double-click to export*	**clk_0**
	reset	Reset Input	*Double-click to export*	[clk]
	s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
	external_connection	Conduit	**otg_hpi_address**	
	otg_hpi_data	PIO (Parallel I/O) Intel FPGA IP		
	clk	Clock Input	*Double-click to export*	**clk_0**
	reset	Reset Input	*Double-click to export*	[clk]
	s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
	external_connection	Conduit	**otg_hpi_data**	
	otg_hpi_r	PIO (Parallel I/O) Intel FPGA IP		
	clk	Clock Input	*Double-click to export*	**clk_0**
	reset	Reset Input	*Double-click to export*	[clk]
	s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
	external_connection	Conduit	**otg_hpi_r**	
	otg_hpi_w	PIO (Parallel I/O) Intel FPGA IP		
	clk	Clock Input	*Double-click to export*	**clk_0**
	reset	Reset Input	*Double-click to export*	[clk]
	s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
	external_connection	Conduit	**otg_hpi_w**	

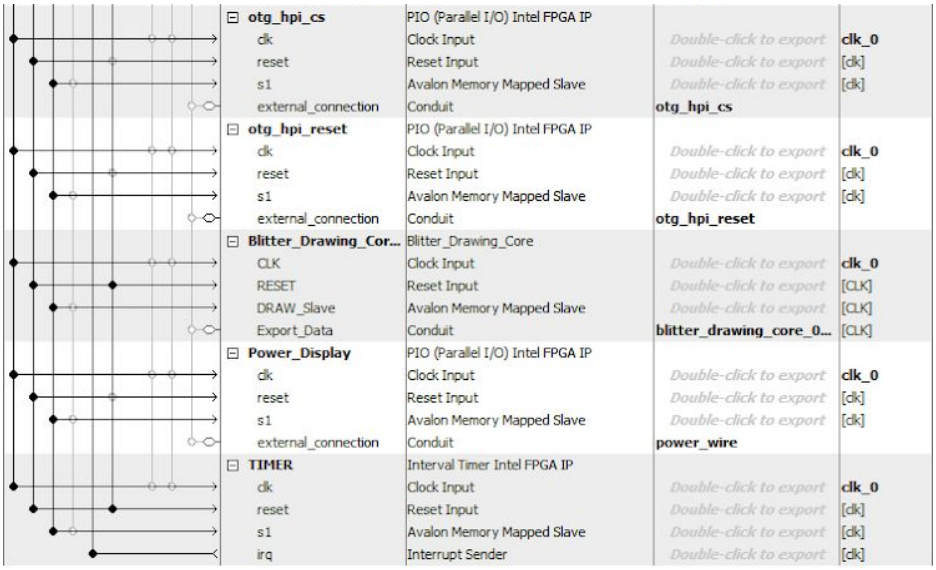


Figure 20 Top level of Qsys for NIOS II SOC

4.4.2. Qsys Modules

Module: nios2_gen2_0

Exports:

Description:

This is an economy version processor we allocate for NIOS II. It handles all the logical operations and computations our program might use. One more thing to notice is that this processor also supports JTAG Debug, which means we can debug using "printf" in Eclipse.

Module: onchip_memory2_0

Exports:

Description:

This is the onchip_memory module for our system, which is always used as cache or data buffer. On-chip memory is relatively "expensive" since it required a lot more logic elements, but it can significantly improve the memory operation speed compared on external memory chips.

Module: sdram

Exports: sdram_wire

Description:

This is the SDRAM controller for NIOS II CPU. SDRAM need to be refreshed every time as we do memory operations, we don't need to write a driver by ourselves, because this existing SDRAM IP module can be used in this case.

Module: sdram_pll

Exports: sdram_clk

Description:

Pll stands for "Phase Lock Loop." As suggested by the name, it's used to generate a second clock signal shifted by an amount. As explained in detail in the following subsection(INQ Question), this block is necessary to prevent skew operation.

Module: sysid_qsys_0
Exports:
Description:

This system ID checker is to ensure the compatibility between hardware and software. This module will give us a serial number, which the software loader checks against when we start the program. This prevents us from loading software onto an FPGA which has an incompatible NIOS II configuration.

Module: jtag_uart_0
Exports:
Description:

This module is used to let the FPGA board communicate with the PC. Thus the "scanf" and "printf" function in C program can be effectively implemented, and we can send data as well as read the result to/from the FPGA board.

Module: keycode
Exports: keycode
Description:

This is a PIO block to show the keycode pressed on the hex display. At most two keys pressed at the same time can be at the keycode module, then the hex display can display two key's keycode.

Module: otg_hpi_address
Exports: otg_hpi_address
Description:

This is a 2-bit signals representing the address that should be accessed on the EZ-OTG chip. The signals come out of the software and will be parsed to connect with the actual address pins on the EZ-OTG chips. Details of the addresses are listed as follows:

Port Registers	HPI A [1]	HPI A [0]	Access
HPI DATA	0	0	RW
HPI MAILBOX	0	1	RW
HPI ADDRESS	1	0	W
HPI STATUS	1	1	R

Table 2 HPI Interface Addresses

Module: otg_hpi_data
Exports: otg_hpi_data
Description:

This is a 16-bit signals representing the data to be written to or read from the EZ-OTG chip. Therefore, this is an inout port. The signals that come out of the software will be parsed to connect with the actual address pins on the EZ-OTG chips.

Module: otg_hpi_r
Exports: otg_hpi_r
Description:

This is a 1-bit signal representing the read signal on the EZ-OTG chip memory. The signal comes out of the software and will be parsed to connect with the actual address pins on the EZ-OTG chips.

Module: otg_hpi_w
Exports: otg_hpi_w
Description:

This is a 1-bit signal representing the write signal on the EZ-OTG chip memory. The signal comes out of the software and will be parsed to connect with the actual address pins on the EZ-OTG chips.

Module: otg_hpi_cs
Exports: otg_hpi_cs
Description:

This is a 1-bit signal representing the chip select signal on the EZ-OTG chip memory. The signal comes out of the software and will be parsed to connect with the actual address pins on the EZ-OTG chips.

Module: otg_hpi_reset
Exports: otg_hpi_reset
Description:

This is a 1-bit signal representing the reset signal on the EZ-OTG chip memory. The signal comes out of the software and will be parsed to connect with the actual address pins on the EZ-OTG chips.

Module: Blitter_Draw_Core
Exports: blitter_drawing_core_0_export_data
Description:

This module is our custom IP core based on the avalon_draw_interface.sv. Thus, the registers in the interface can be memory mapped to the NIOS II system. It provides us a more direct way to access and operate the registers values in the C program. So Draw.c can control what we are going to display on the screen.

Module: Power_Display
Exports: power_wire
Description:

This is a PIO block to show the current power level of the cannon ball. It has 8 levels, and would be shown on the Green LEDs.

Module: TIMER
Exports:
Description:

This module is used for the time limit function in our game. The interval timer can send periodic interrupts. So the system could know when should the game stop.

5. Software Explanation

5.1. Specific.h

To clarify our code and facilitate debugging and reconstruction, we keep every constant parameter in this file as a macro. Concretely, we define AVALON interface related parameters, image pattern related specifics, game logic related specifics, and some helper functions.

```
#pragma once

#define PI 3.1415926
//Properties
/**********AVALON INTERFACE RELATED*********/
#define IDX DRAW_PTR[0]
#define X_START DRAW_PTR[1]
#define Y_START DRAW_PTR[2]
#define X_STOP DRAW_PTR[3]
#define Y_STOP DRAW_PTR[4]
#define FRAME_DONE DRAW_PTR[5]
#define DRAW_START DRAW_PTR[6]
#define DRAW_DONE DRAW_PTR[7]
static volatile unsigned int * DRAW_PTR = (unsigned int *) 0x00000020;
static volatile unsigned int * POWER_PTR = (unsigned int *) 0x00000040;
```

Figure 21 Avalon Interface Related Definitions

Drawing specifics include the width and height of the image pattern, VGA X and Y locations of fixed objects, point assigned to each fish, length of the animation frames of each object, the start of the pattern index provided to the IndMatcher described above, and so forth.

```
#define FLATFISH_WIDTH 111
#define FLATFISH_HEIGHT 42
#define FLATFISH_PT 8
#define FLATFISH_LEN 5
#define FLATFISH_IDXSTART 81

#define BLUFISH_WIDTH 34
#define BLUFISH_HEIGHT 25
#define BLUFISH_PT 3
#define BLUFISH_LEN 4
#define BLUFISH_IDXSTART 91

#define CANNON_WIDTH 212
#define CANNON_HEIGHT 108
#define CANNON_IDXSTART 1
#define CANNON_X 214
#define CANNON_Y 372

#define SUPPORT_WIDTH 70
#define SUPPORT_HEIGHT 40
#define SUPPORT_IDXSTART 24
#define SUPPORT_X 290
#define SUPPORT_Y 440

#define NET_WIDTH 101
#define NET_HEIGHT 84
#define NET_LEN 4
#define NET_IDXSTART 25
#define NET_RADIUS_SQR  2139
```

Figure 22 Examples of Image Pattern Related Definitions

Game logic related specifics include the minimum and maximum number of each type of fish, min and max velocity of each type of fish, velocity of cannon ball, and so forth.

```
#define NUM_TYPE_FISH 6
//TRAJECTORY FUNCTION
#define FREQ_MIN 5
#define FREQ_MAX 10
#define AMP_MIN 10
#define AMP_MAX 20
//FISH NUMBER AND VELOCITY
#define BW_FISH_NUM_MIN 4
#define BW_FISH_NUM_MAX 7
#define BW_FISH_VELO_MIN 2
#define BW_FISH_VELO_MAX 4

#define P_FISH_NUM_MIN 3
#define P_FISH_NUM_MAX 5
#define P_FISH_VELO_MIN 3
#define P_FISH_VELO_MAX 5

#define Y_FISH_NUM_MIN 8
#define Y_FISH_NUM_MAX 15
#define Y_FISH_VELO_MIN 5
#define Y_FISH_VELO_MAX 10
```

Figure 23 Examples of Game Logic Related Definitions

5.2. *Struct.h

Due to the large number of varying structures appearing in our game, we create lots of struct to assist our design. Concretely, we have five distinct structs in our code, which are Animal, Cannon, CannonBall, FishNet, and GameState. The meaning of each struct is suggested by its name, except for the GameState, which represents the state of a single game played. It contains the total score earned by the player, stop time, and a variable indicating whether the game is naturally stopped or forcibly stopped.

```
typedef struct animalStruct{
    int point;
    int curIdx; //Current index in the animation frames
    int idxStart;
    int* pat;
    int len;    //Length of the frames(GIF)
    int x;
    int y;
    int width;
    int height;
    int velocity;   //Pos means going right, vice versa
    int func;
    int caught;
    int caughtFrame;
}Animal;
```

```
typedef struct CannonBallStruct{
    int curIdx;
    int x;
    int y;
    int hidden;
    int hit;
    int deg;
    int fix;
    int frame;
    float distance;
    int stopY;
    int x_velo;
    int y_velo;
}CannonBall;
```

```
typedef struct CannonStruct{
    int degree;     //Degrees from the left horizontal
    int idxStart;
    int curIdx;
}Cannon;
```

```
typedef struct fishNetStruct{
    int curIdx;
    int x;
    int y;
    int hidden;
    int frame;
}FishNet;
```

```
typedef struct GameStateStruct{
    int time;   //How much time player has played
    int score;
    int state;  //If 0, naturally stop. //If 1, forced stop
}GameState;
```

Figure 24 Examples of Structs

5.3. *Prototype.h

Creating prototypes is common in software engineering, and we also use it to quickly populate objects and promptly shift states of an object. Each prototype corresponds to the object of initial state, and we can use it reset an object. For example, we use the fish prototype to quickly generate multiple fishes and only modify certain values that need to be randomized. We also use FishNet and CannonBall prototypes to swiftly shift objects between visible and hidden states as well as busy and idle states.

```
static Animal BW_FISH = {.pat = BWFISH, .len = BWFISH_LEN,\
        .x = 610, .y = 200, .width = BWFISH_WIDTH,\
        .height = BWFISH_HEIGHT, .point = BWFISH_PT,.curIdx = BWFISH_IDXSTART,\
        .idxStart = BWFISH_IDXSTART, .velocity = 3, .caught = 0, .caughtFrame = 0};

static Animal P_FISH = {.pat = PFISH, .len = PFISH_LEN, .x = 15, .y = 70,\
        .width = PFISH_WIDTH, .height = PFISH_HEIGHT, .point = PFISH_PT,\
        .curIdx = PFISH_IDXSTART, .idxStart = PFISH_IDXSTART, .velocity = 3, .caught = 0, .caughtFrame = 0};

static Animal Y_FISH = {.pat = YFISH, .len = YFISH_LEN, .x = 15, .y = 70,\
        .width = YFISH_WIDTH, .height = YFISH_HEIGHT, .point = YFISH_PT,\
        .curIdx = YFISH_IDXSTART, .idxStart = YFISH_IDXSTART, .velocity = 3, .caught = 0, .caughtFrame = 0};

static Animal FLAT_FISH = {.pat = FLATFISH, .len = FLATFISH_LEN, .x = 15, .y = 70,\
        .width = FLATFISH_WIDTH, .height = FLATFISH_HEIGHT, .point = FLATFISH_PT,\
        .curIdx = FLATFISH_IDXSTART, .idxStart =FLATFISH_IDXSTART, .velocity = 3, .caught = 0, .caughtFrame = 0};

static Animal R_FISH = {.pat = RFISH, .len = RFISH_LEN, .x = 15, .y = 70,\
        .width = RFISH_WIDTH, .height = RFISH_HEIGHT, .point = RFISH_PT,\
        .curIdx = RFISH_IDXSTART, .idxStart = RFISH_IDXSTART, .velocity = 3, .caught = 0, .caughtFrame = 0};

static Animal BLU_FISH = {.pat = BLUFISH, .len = BLUFISH_LEN, .x = 15, .y = 70,\
        .width = BLUFISH_WIDTH, .height = BLUFISH_HEIGHT, .point = BLUFISH_PT,\
        .curIdx = BLUFISH_IDXSTART, .idxStart = BLUFISH_IDXSTART, .velocity = 3, .caught = 0, .caughtFrame = 0};
```

Figure 25 Examples of Prototypes

5.4. Main.c

The main function contains the menu and game state transition logic.

```
int main(){
    keyboard_init();
    srand((unsigned)time(NULL));    //INIT SEED
    char option;
    int scores[5] = {0,0,0,0,0};
    MENU:
    option = displayMenu();
    if(option == 'p'){
        GameState gamestate = playGame();
        updateScores(scores, gamestate.score);
        goto MENU;
    }
    else if(option == 's'){
        displayScores(scores);
        goto MENU;
    }
}
```

Figure 26 Code of Main.c

As the system starts, it would to finish the preparation first, which contains keyboard initialization, random seed initialization, and score records initialization. Then it would display the menu with “displayMenu()”, and wait for the option input. If option “p” is selected as the user put the arrow to the “Start” icon and press enter, the game would start. After 60 seconds or the user’s manual interrupt, the game would be stopped and the score would be updated. The user would be brought back to the menu for the next round or viewing the high score records. If option “s” is selected as the user put the arrow to the “High Scores” icon and press enter, high scores table would be displayed, containing 5 top scores. The user can press “Esc” to interrupt and go back to the menu.

5.5. MainLogic.c

"MainLogic.c" contains functions for the procedure of playGame(), displayMenu(), updateScores(), and displayScores().

5.5.1. playGame()

"playGame" defines the procedure for the game logic. Before the loop, it initializes fishes, the cannon, the cannonball, the fishnet and the background. It would also record the game start time, and this would be used in check with the time limit.

```
GameState playGame(){
    int total;      //Total number of fish
    int nums[NUM_TYPE_FISH];    //Number of each type of fish
    int power_use = 0;
    int power_calc = 0;
    int key = 0;
    int prevkey = 0;
    int total_score = 0;
    int prev_total = 0;
    int msec = 0;
    int prevsec = 0;
    int sec = 0;
    randomNumbers(nums, &total);    //generate necessary random numbers
    Animal animals[total];      //Array that store animals
    generateFish(nums, animals);    //Generate fish randomly
    Cannon cannon = {.degree = 90, .curIdx = 12, .idxStart = CANNON_IDXSTART};
    CannonBall cannonball = CANNON_BALL_PRO;
    FishNet fishnet = FISH_NET_PRO;
    init_background();  //Initialize background
    printf("Start Gaming\n");
    FRAME_DONE = 1;
    int flag = 0;   //If flag is 0, naturally stop. If flag is 1, forced stop
    clock_t before = clock();
```

Figure 27 playGame() part A

Then the program would enter a loop, and each loop means a frame. In each loop, the program would call functions defined in GameLogic.c, like move_connon(), updateFishNet(), updateCannonBall(), and updateFish() which would change the behavior of each object on the display.

There are two ways to exit this loop. The first way is pressing the "Esc" key, and the second way is exceeding the time limit, which we set as 60 seconds, then it would break the loop and finish executing the playGame function.

```
while(1){
    prevkey = key;
    key = get_key();
    if(key == 41){
        flag = 1;
        break;
    }
    power_calc = keyPress(key,power_calc);
    move_cannon(&cannon, key);
    updateFishNet(&fishnet, cannonball);
    updateCannonBall(cannon.degree, prevkey, key, power_use, &cannonball);
    showpower(power_use);
    power_use = power_calc;
    draw_frame(total,animals,cannon,cannonball,fishnet,total_score, 60-sec,0);
    delay_loop();
    updateFish(total, animals, fishnet, &total_score);
    prev_total = total_score;
    clock_t difference = clock() - before;
    msec = difference * 1000 / CLOCKS_PER_SEC;
    sec = msec/1000;
    if(sec > 60){
        int frame_count = 1;
        for(int i = 0; i < 7; i++){
            draw_frame(total,animals,cannon,cannonball,fishnet,total_score, 60-sec,frame_count);
            frame_count++;
            setTimeout(500);
        }
        break;
    }
    prevsec = sec;
}
```

Figure 28 playGame() part B

5.5.2. displayMenu()

"displayMenu" is similar to "draw_frame", but more simpler. It displays the background, the logo and options statically, while the arrow could be moved to two positions determined by the keyboard. Once enter is entered, displayMenu would return the option value "p" or "s" which represents the user's selection.

```
char displayMenu(){
    init_background();
    draw_Logo();
    draw_Options();
    int key;
    int pos = 0;
    char option;
    while(1){
        FRAME_DONE = 0;
        init_background();
        draw_Logo();
        draw_Options();
        key = get_key();
        if(pos == 0 && key == 81)
            pos = 1;
        else if(pos == 1 && key == 82)
            pos = 0;
        draw_Arrow(pos);
        FRAME_DONE = 1;
        delay_loop();
        if(key == 40){
            option = pos ?'s':'p';
            break;
        }
    }
    return option;
}
```

Figure 29 Code of displayMenu()

5.5.3. updateScores()

Our game keeps tracking the 5 top scores. When a new score comes in, it would be sorted into our high score array.

```
void updateScores(int* scores, int cur_score){
    for(int i = 0; i < 5; i++){
        if(cur_score > scores[i]){
            int temp = scores[i];
            int temp2;
            scores[i] = cur_score;
            for(int j = i+1; j < 5; j++){
                temp2 = scores[j];
                scores[j] = temp;
                temp = temp2;
            }
            return;
        }
    }
}
```

Figure 30 Code of updateScores()

5.5.4. displayScores()

"displayScores" is used for displaying the top 5 scores, and its functionality is drawing a static frame. Because the numbers are stored in our sprite sheet as single digit, so we need to specifically calculate each digit's value for all 5 scores. When the "Esc" key is pressed, the displayScores() would quit.

```
void displayScores(int* scores){
    init_background();
    int key = 0;
    IDX = HIGHSCORE_IDXSTART;
    X_START = 320-HIGHSCORE_WIDTH/2;
    Y_START = 120-HIGHSCORE_HEIGHT/2;
    X_STOP = X_START + HIGHSCORE_WIDTH;
    Y_STOP = Y_START + HIGHSCORE_HEIGHT;
    DRAW_START = 1;
    while(!DRAW_DONE);
    DRAW_START = 0;
    for(int i = 1; i<=5; i++){
        draw_Num(i,280, 160 + i * 35);
        draw_SC(282+NUMBER_WIDTH, 160+i*35);
        int digit3 = scores[i-1]/100;
        if(digit3 == 0)
            digit3 = 10;
        int digit2 = scores[i-1]/10%10;
        if(digit3 == 10 && digit2 == 0)
            digit2 = 10;
        int digit1 = scores[i-1]%10;
        draw_Num(digit3,318+NUMBER_WIDTH,160+i*35);
        draw_Num(digit2,320+NUMBER_WIDTH*2,160+i*35);
        draw_Num(digit1,322+NUMBER_WIDTH*3,160+i*35);
    }
    while(1){
        key = get_key();
        if(key == 41){
            break;
        }
    }
}
```

Figure 31 Code of displayScores()

5.6. GameLogic.c

This file provides all the game logic related helper functions that will be used in our MainLogic.c.

Functions (contained in header file),

```
void randomNumbers();
void generateFish();
void updateFish();
int func1();
int func2();
int isCaught();
void move_cannon();
void updateCannonBall();
void updateFishNet();
float maxLength();
int keyPress();
void showpower();
void setTimeout();
```

Figure 32 Code of GameLogic.h

5.6.1. randomNumbers()

This function will call the random number generator and provide the necessary random numbers to the generateFish function for randomized fish generation. We simply use the random generator provided in C and limit it to a given range constrained by NUM_MIN and NUM_MAX. We then record the total number and pass it to the upper level to create an array that holds the fish objects.

```
void randomNumbers(int* nums, int* total){
    int BWNUM = rand() % (BW_FISH_NUM_MAX - BW_FISH_NUM_MIN + 1) + BW_FISH_NUM_MIN;
    int PNUM = rand() % (P_FISH_NUM_MAX - P_FISH_NUM_MIN + 1) + P_FISH_NUM_MIN;
    int YNUM = rand() % (Y_FISH_NUM_MAX - Y_FISH_NUM_MIN + 1) + Y_FISH_NUM_MIN;
    int RNUM = rand() % (R_FISH_NUM_MAX - R_FISH_NUM_MIN + 1) + R_FISH_NUM_MIN;
    int FLATNUM = rand() % (FLAT_FISH_NUM_MAX - FLAT_FISH_NUM_MIN + 1) + FLAT_FISH_NUM_MIN;
    int BLUNUM = rand() % (BLU_FISH_NUM_MAX - BLU_FISH_NUM_MIN + 1) + BLU_FISH_NUM_MIN;
    nums[0] = BWNUM;
    nums[1] = PNUM;
    nums[2] = YNUM;
    nums[3] = RNUM;
    nums[4] = FLATNUM;
    nums[5] = BLUNUM;

    *total = BWNUM + PNUM + YNUM + RNUM + FLATNUM + BLUNUM ;
}
```

Figure 33 Code of randomNumbers()

5.6.2. generateFish()

This function will create a randomized number of fish according to the numbers generated in randomNumbers. We initialize each fish randomly based on its base prototype. As explained in the above sections, its x, y, velocity, and moving trajectory will all be randomized.

```
void generateFish(int* nums, Animal * animals){
    //First part corresponds to BW_FISH
    //Second part corresponds to P_FISH
    int idx = 0;
    for(int i = 0; i < NUM_TYPE_FISH; i++){
        int curNum = nums[i];
        switch(i){
        case 0 : //BW_FISH
            for(int j = 0; j < curNum; j++){
                Animal bwfish = BW_FISH;
                bwfish.y = rand() % (480 - BWFISH_HEIGHT);
                bwfish.x = rand() % (640 - BWFISH_WIDTH);
                bwfish.velocity = -1 * (rand() % (BW_FISH_VELO_MAX - BW_FISH_VELO_MIN + 1) + BW_FISH_VELO_MIN);
                bwfish.func = rand() % 2;
                animals[idx + j] = bwfish;
            }
            break;
        case 1: //P_FISH
            for(int j = 0; j < curNum; j++){
                Animal pfish = P_FISH;
                pfish.y = rand() % (480 - PFISH_HEIGHT);
                pfish.x = rand() % (640 - PFISH_WIDTH);
                pfish.velocity = (rand() % (P_FISH_VELO_MAX - P_FISH_VELO_MIN + 1) + P_FISH_VELO_MIN);
                pfish.func = rand() % 2;
                animals[idx + j] = pfish;
            }
            break;
        case 2: //Y_FISH
            for(int j = 0; j < curNum; j++){
                Animal yfish = Y_FISH;
                yfish.y = rand() % (480 - YFISH_HEIGHT);
                yfish.x = rand() % (640 - YFISH_WIDTH);
                yfish.velocity = -1 * ((rand() % (Y_FISH_VELO_MAX - Y_FISH_VELO_MIN + 1) + Y_FISH_VELO_MIN));
```

Figure 34 Code of generateFish()

5.6.3. isCaught()

This function returns an integer indicating whether the given fish is captured by the fishnet. The detailed explanation is provided in the above section.

```
int isCaught(FishNet fishnet, Animal animal){
    float cent_X_FishNet = fishnet.x + NET_WIDTH/2.0;
    float cent_Y_FishNet = fishnet.y + NET_HEIGHT/2.0;
    float cent_X_animal = animal.x + animal.width/2.0;
    float cent_Y_animal = animal.y + animal.height/2.0;

    float distance_sqr = (cent_X_animal - cent_X_FishNet) * (cent_X_animal - cent_X_FishNet)\
          + (cent_Y_animal - cent_Y_FishNet) * (cent_Y_animal - cent_Y_FishNet);
    return ((fishnet.frame == 4) && (distance_sqr < NET_RADIUS_SQR));
}
```

Figure 35 Code of isCaught()

5.6.4. updateFish()

This function will loop through our fish array and update each fish object respectively. Concretely, it will update each fish's animation frame, x and y locations based on its randomized velocity and trajectory, and check whether the fish has been caught. We also pass in the total_score to update it after our collision detection.

```
void updateFish(int total, Animal* animals, FishNet fishnet, int* total_score){
    for(int i = 0; i < total; i++){
        if(animals[i].caught){
            animals[i].caughtFrame++;
            if(animals[i].caughtFrame > 8){
                animals[i].x = 640;
                animals[i].y = rand() % (480 - animals[i].height);
                animals[i].caught = 0;
                animals[i].caughtFrame = 0;
            }
            continue;
        }
        if(!fishnet.hidden){
            if(isCaught(fishnet, animals[i])){
                animals[i].caught = 1;
                *total_score += animals[i].point;
                continue;
            }
        }
        animals[i].x += animals[i].velocity;
        switch(animals[i].func){
        case 0 : //y unchanged
            break;
        case 1:
            //animals[i].y = func1(animals[i].x, animals[i].y);
            break;
        case 2:
            animals[i].y = func2(animals[i]);
            break;
        }
        animals[i].curIdx = ((animals[i].curIdx - animals[i].idxStart+1) % animals[i].len) + animals[i].idxStart;
        if(animals[i].x <= -1*(animals[i].width)){
            animals[i].x = 640;



            animals[i].y = rand() % (480 - animals[i].height);
        }
        if(animals[i].x > 640){
            animals[i].x = -1*(animals[i].width)+3;
            animals[i].y = rand() % (480 - animals[i].height);
        }
        if(animals[i].y > 480){
            animals[i].y = -1*(animals[i].height) + 3;
            animals[i].x = rand() % (640 - animals[i].width);
        }
        if(animals[i].y <= -1*(animals[i].height)){
            animals[i].y = 480;
            animals[i].x = rand() % (640 - animals[i].width);
        }
    }
}
```

Figure 36 Code of isCaught()

5.6.5. Move_cannon()

This function will update the current cannon index and location based upon the keycode passed in. Basically, it realizes the cannon rotation.

```
void move_cannon(Cannon* cannon, int key){
    if(key == 7 || key == 79){
        if(cannon -> curIdx != 1){
            cannon -> curIdx --;
            cannon -> degree -= 8;
        }
    }
    else if(key == 4 || key == 80){
        if(cannon -> curIdx != 23){
            cannon -> curIdx ++;
            cannon -> degree += 8;
        }
    }
}
```

Figure 37 Code of move_cannon()

5.6.6. maxLength()

This function calculates the maximum distance can be traveled by the cannonball based upon the current degree. Detailed analysis of how we come up with this function is provided in the above section.

```
float maxLength(int degree){
    if(degree > 56 && degree < 124)
        return 430.0 / sin(degree / 180.0 * PI);
    else
        return abs(320.0 / cos(degree / 180.0 * PI));
}
```

Figure 38 Code of maxLength()

5.6.7. updateCannonBall()

This function will update the cannon ball at each frame. It realizes the cannonball rotating animation, updates the cannonball x and y locations based on its velocity, and check whether it reaches the stop location.

```
void updateCannonBall(int deg, int prevkey, int key, int power, CannonBall* cannonball){
    if(prevkey == 44 && key != 44 && cannonball -> fix == 0){
        cannonball -> hidden = 0;
        //cannonball -> deg = deg;
        cannonball -> fix = 1;
        cannonball -> distance = maxLength(deg)/8.0 * power;
        cannonball -> stopY = 430 - ((cannonball -> distance) * sin(deg / 180.0 * PI));
        cannonball -> x_velo = CANNON_BALL_VELO * cos(deg / 180.0 * PI);
        cannonball -> y_velo = -1 * sin( deg / 180.0 * PI) * CANNON_BALL_VELO;
        //printf("stopY: %d\n",cannonball -> stopY );
    }

    if(!(cannonball -> hidden)){
        //x_velo = CANNON_BALL_VELO * cos(degree / 180.0 * PI);
        //y_velo = -1 * sin( degree / 180.0 * PI) * CANNON_BALL_VELO;

        if(cannonball->y <= cannonball->stopY)
            *cannonball = CANNON_BALL_PRO;

        cannonball -> x += cannonball -> x_velo;
        cannonball -> y += cannonball -> y_velo;
        cannonball -> curIdx = ((cannonball -> curIdx - CANNON_BALL_IDXSTART+1) % CANNON_BALL_LEN) + CANNON_BALL_IDXSTART;
        cannonball -> frame ++;

        if(cannonball -> x > 640 || cannonball -> x <= -1*CANNON_BALL_WIDTH || cannonball -> y > 480 || cannonball -> y <=
            -1*CANNON_BALL_HEIGHT){
            *cannonball = CANNON_BALL_PRO;
        }
    }
}
```

Figure 39 Code of updateCannonBall()

5.6.8. updateFishNet()

This function checks whether the cannonball has reached the destination and update fishnet to visible if it does. Moreover, it achieves the largening animation effects of the fishnet.

```
void updateFishNet(FishNet* fishnet, CannonBall cannonball){
    if(cannonball.y <= cannonball.stopY){
        fishnet -> hidden = 0;
        fishnet -> x = cannonball.x + CANNON_BALL_WIDTH/2 - NET_WIDTH/2;
        fishnet -> y = cannonball.y + CANNON_BALL_HEIGHT/2 - NET_HEIGHT/2;
    }
    if(!fishnet -> hidden){
        fishnet -> curIdx += 1;
        if(fishnet -> curIdx > 28)
            fishnet -> curIdx = 28;
        fishnet -> frame ++;
        if(fishnet -> frame > 10)
            *fishnet = FISH_NET_PRO;
    }
}
```

Figure 40 Code of updateFishNet()

5.6.9. func1() & func2()

These functions provide trajectory for fish to travel in the ocean environment.

```
int func1(Animal animal){
    //double freq =  0.01 * (rand() % (FREQ_MAX - FREQ_MIN + 1) + FREQ_MIN);
    //int amp = rand() % (AMP_MAX - AMP_MIN + 1) + AMP_MIN;
    return 10*sin(0.08*animal.x) + animal.y;
}

int func2(Animal animal){
    return animal.y + animal.velocity / 3 * 2;
}
```

Figure 41 Code of func1() & func2()

5.7. Draw.c

Functions (contained in header file):

```
void init_background();
void draw_frame();
void draw_animal();
void draw_cannon();
void draw_CannonBall();
void draw_Logo();
void draw_Options();
void draw_Arrow();
void draw_Score_Ingame();
void draw_Time_Ingame();
void draw_SC();
```

Figure 42 Code of Draw.h

Purpose & Description:

Draw.c is responsible for drawing all the patterns on the screen. In each draw single element functions, the software would write the pattern information to the avalon draw interface, and wait for the Draw Done signal. In draw multiple element function, draw_Frame(), the software executes draw single element functions in a proper sequence and control the Frame Done signal for sync the second frame buffer.

Draw single element function example:

```
void draw_animal(Animal animal){
    IDX = animal.curIdx;
    X_START = animal.x;
    Y_START = animal.y;
    X_STOP = animal.x + animal.width;
    Y_STOP = animal.y + animal.height;
    DRAW_START = 1;
    while(!DRAW_DONE);
    DRAW_START = 0;
}
```

Figure 43 Code of draw_animal()

When drawing a fish, we need to call the “draw_animal” function. We pass the animal which need to be drawn to the function, so the function can know the index of pattern, the start X and Y position and the end X and Y position of the pattern on the screen, and write these values to the avalon draw interface. When the software finishes sending these values, it would toggle the “DRAW_START” label to “1”. The hardware blitter keeps monitoring the

"DRAW_START" label, so it can know when to start drawing the pattern. If the hardware finishes drawing the pattern, it would provide a feedback to the software called "DRAW_DONE". Once the software gets the "DRAW_DONE" signal, it would move on and reset the "DRAW_START" to 0.

Draw multiple elements function:

```
void draw_frame(int num, Animal animals[], Cannon cannon, CannonBall cannonball,
    FishNet fishnet, int score, int sec, int frame_count){
    FRAME_DONE = 0;
    init_background();
    if(sec >= 0 && sec <= 60){
        draw_CannonBall(cannonball);
        draw_cannon(cannon);
    }
    for(int i = 0; i < num; i++){
        if(!(animals[i].caughtFrame % 2))
            draw_animal(animals[i]);
    }
    if(sec >= 0 && sec <= 60){
        draw_FishNet(fishnet);
    }
    if(sec >= -1 && sec <= 60 &&(!(frame_count % 2))){
        draw_Score_Ingame(score);
        sec = (sec < 0) ? 0 :sec;
        draw_Time_Ingame(sec);
    }
    FRAME_DONE = 1;
}
```

Figure 44 Code of draw_frame()

"draw_frame" is the function that draws all elements within a frame. It takes the number of fishes, the fishes array, the cannon, the cannonball, the fishnet, the score, the second and the frame count, and they are the information that would be displayed. First, the software would toggle the "Frame Done" off to 0, so the double buffer would not sync anymore. This can guarantee we would not display an unfinished frame, and prevent elements from blinking. Then, the software would do init_backgroud(), which would cover all elements drawn in the previous frame. After the background has been initiated, the software would start to draw different layers of our game scene, the cannonball layer, cannon layer, fishes layer, net layer and characters layer from backwards to forwards. We design the layers in this way in order to make sure everything can be shown appropriately even though they may overlap with each other. For example, when a fishnet catches a fish, the fishnet should be on the top of the fish. In order to do that, we need to draw the fish first in the frame, and then draw the fishnet. For the fishes and the net, they have blinking effect before they vanish from the screen. We achieve this by adding a counter for them, when the counter is an odd number, the fishes and the net would not be drawn. After drawn every thing in the frame, the software would toggle "Frame Done" to 1, so the hardware is able to sync two frame buffers, and the completed scene can be displayed on the screen.

5.8. Keyboard.c

"Keyboard.c" is responsible for the keyboard setup. We use the keyboard controller from Lab 8. However, the software is designed with polling IO, and the original design is checking the pressed key in an infinite loop. Thus, we split the original keyboard controller into keyboard_init() and get_key().

6. Design Resources and Statistics

LUT	5039
DSP	0
Memory (BRAM)	3083392 bits
M9Ks	382
SRAM	2.1MB
PLL	3
Flash	1.31MB
Flip-Flop	2889
Frequency	55.37MHz
Static Power	108.79mW
Dynamic Power	70mW
Total Power	303.17mW

7. Conclusion

During the final project, we collaborated perfectly and completed the project up to our expectations. We learned many useful techniques such as implementing blitter and double buffer in FPGA to produce smooth and stable visual display. Our ability to write in C and coordinate software and hardware to work as a whole also enhances. Decorating the project with a background music gives us the opportunity to learn how I2C and the audio chip operates on the DE2 board and will be of tremendous help for us to deal with audio-related projects in the future. At the beginning, we proposed to make a motion control game and start by trying to make the camera module to work. However, due to the hardware(defected camera modules) limitation, we promptly shift to only design and build the game and try to improve our game as much as

possible. It proves that we made the correct decision at that time since we're currently pretty satisfied with our accomplishments. We finished all of the basic and extra points in the proposal and add even more fancy features. During the design process, we encountered numerous problems and difficulties, including things like preventing the screen from blinking, configuring the audio driver, debugging C program, etc. Simulation tool like ModelSim Altera assists us greatly to debug and overcome the challenge. Also, we always think out of the box and try to find a more superior approach than the current one, which will lead to better user experience. For example, we ponder over the user control for a long time and eventually determines that the current implementation would be the most attractive. We then add LEDs display to give the user a better visual experience. However, we think we can still improve the current version. For example, we can a double player mode for two players to compete at the same time. Moreover, some additional hardware control approach, like using flex sensors, would be very intriguing as well.