ECE 385
Fall 2017

Implementation of Feeding Frenzy in System Verilog

Anh Leu/ Michelle Lamblin
Section ABE/Tuesday 11 a.m. – 2 p.m.
TA: Zhenhong Liu, Gene Shiue

Table of Contents

Introduction

Our project designs and creates an implementation of the game Feeding Frenzy, a game in which a fish consumes other fish smaller than itself to grow larger to eat fish of a larger size. We decided that this project would be an effective use of our skills and enjoyable to make. This project will expand our knowledge and skills of the FPGA. The result was an enjoyable game with several characters and many functionalities applied from our knowledge obtained in ECE385 through our lectures and projects throughout the semester.

Figure 1: Original Feeding Frenzy game

The portion of the game that we are implementing is the survival mode from the game Feeding Frenzy, the arcade-style survival mode written by Sprout Games which was released in 2004. The objective of the game is to grow larger by consuming fish that are smaller than you are and survive if possible before being eaten by other characters. The version we have made features our professor, Zuofu Cheng (with his permission), as the bonus power up in the game along with a jellyfish which reset the points to 0 and revert our main character to its lowest level if touched.

Figure 2: Professor Zuofu

Figure 3: jellyfish

Description of Circuit

Our final project circuit has many different circuit components and files. The .sv files setup the circuitry and wired connections between the current inputs and outputs of the wires. Each component of the .sv files in our code perform a different task within the program and are all used to create the game. The game will be implemented on the FPGA and utilizes a .sv file to control the movements of the fish (Ball.sv), a .sv file to control the timing for the VGA controller (VGAcontroller.sv), and a color mapper to determine the RGB colors of the screen output based on the current location of the on-screen characters. We will implement a EZ-OTG protocol using the cypress CYC67200 chip to interface between the keyboard and the NIOS II to control the character using the keyboard keys up, down, left, and right (x4F, x50, x51, x52).

The keyboard inputs and randomly generated characters will come from C code on the FPGA's version of eclipse and the interface between the FPGA, the keyboard, VGA monitor will be created in the hardware on QSYS. The goal of this is to create a character that can consume other characters smaller than its own size while avoiding potential death from characters larger than itself. Other characters will be randomly generated at the side of the screens and will move horizontally across the player's view. Players can use a keyboard to control the character movements and have it displayed on a VGA monitor.

For the circuit game, it is centered around the sprite main character we developed named *Jodi*. Jodi consumes other characters that she collides with on the screen, and her purpose is to consume smaller fish and avoid larger fish and enemies that are encountered.

We created all the sprites used in this game and they are loaded to the screen using the SRAM from the FPGA board. The game's screen is 640x480 pixels. Characters are randomly generated using our C codes on Eclipse will be passed to hardware part using a PIO block in QSYS. The information on the generated fish will be passed into one of our register. The load patterns were also randomized using C code. Our game.sv module will then analyze the data in the register and decide on the fish data that are at any position on the screen. Any small fish sprite that meets a big fish sprite or any fish that hit the end of the screen (x >= 640 + 32 or x less than or equal to width of the screen along with the largest half width of the fish) will be considered "dead." The register that contains that fish data will be cleared in the next clock cycle and will be reloaded as a different random character at another random position on the screen's edge.

There are a total number of 20 sprites used to create the game and each character is randomly generated and contains a different speed when loaded onto the screen. The sprites generated from the hex file were created writing a python script that fed into a C code we wrote to sequentially convert an array of colors using the 888-color standard (32 bits including 8-bit xFF in front) to an array of colors in 565 standards (16 bits) by bit shifting the colors then write the values in hex

format into a file that can be loaded onto the SRAM. We decided to use the software that were provided from lab 6 to upload the file to the SRAM since we already partitioned our SD card. However, we then realized that the software only takes in files in binary format, so we had to change our C code. However, since C code does not have a built-in function for transferring number to binary with padded 0 bits like in C++, we created a small function with some defined look up to transfer a hex value to binary and write that a HEX file that can be uploaded to FPGA. However, it took a long time to transfer the data, so we decided to use another software that accepted HEX file with hex values and upload the values right after several uploading attempts

.

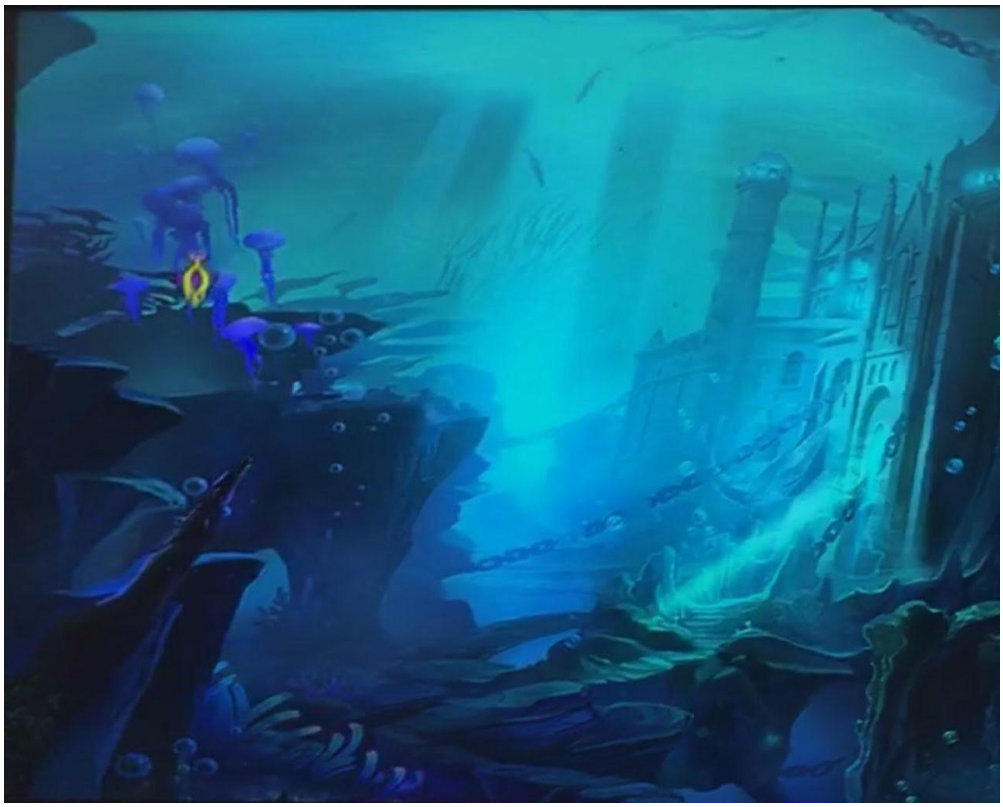


Figure 4: Jodi Loaded from on-chip memory

At first, we wanted to use flash for our character and the background as from SRAM. This allows us to easily have both the fish and the background at the same time. However, this way did not work anymore when we include all our characters due to the memory limitation. We could have reduced the bits using a color palette and able to fit all the sprites to the flash memory. However, at the time we decided to focus on getting the game working then create another clock with higher frequency using QSYS like for SDRAM. We know that we can also compress the data into 8 bits and create a color palette. This can help us get two pixels values at the same time. Which means we will not need to modify the clock but can store the value for next iteration and fetch the background value during the next clock cycle of the current pixel instead. In the end, we did not have enough time to implement that. Instead, we modified the

background picture to become the starting image for the start and end state of our game. We used a deep blue with gradient by Draw Y value to represent the deep ocean that gets darker as we get deeper into the sea. To get the values from the SRAM for the sprites. We first created an SRAM controller in QSYS. However, it turned out to be more complicated than the way we used it in in Lab6, so we decided to implement SRAM that way instead.

The score for the game and the current number of lives is displayed on the FPGA hex displays and display the current score in hex values. When Jodi consumes a fish, a point gets added to her overall score, likewise, if Jodi gets eaten or is killed by another character in the game, she loses a life and starts in the middle of the screen again. All the fish are refreshed. Jodi was designed to start out with 10 lives within the survival mode implementation we created. We decided at first to start with 3 lives like in the original game. However, it turned out to be too hard to get to bigger fish values since it is much harder to control the fish using keyboard comparing to using a mouse. We also decided to not have Jodi speed to be too fast, especially compared to smaller fish and medium fish so it is much harder to gain points, but it is more interesting and challenging to play.

Since we have four instances of Jodi (left, right, up down), we also needed to put the keycode into a D-FF to keep the keycode value to decide on the SRAM address to fetch the correct fish even when no keycode or other keycode is pressed.

Figure 5: Current Key Pressed : Current Lives : Current scores

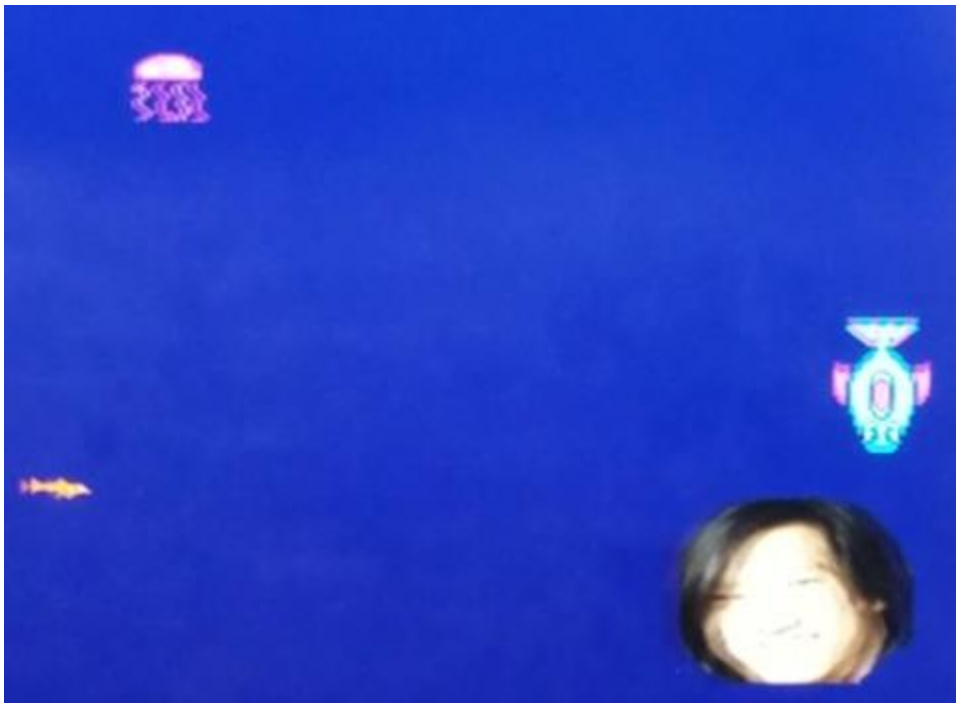
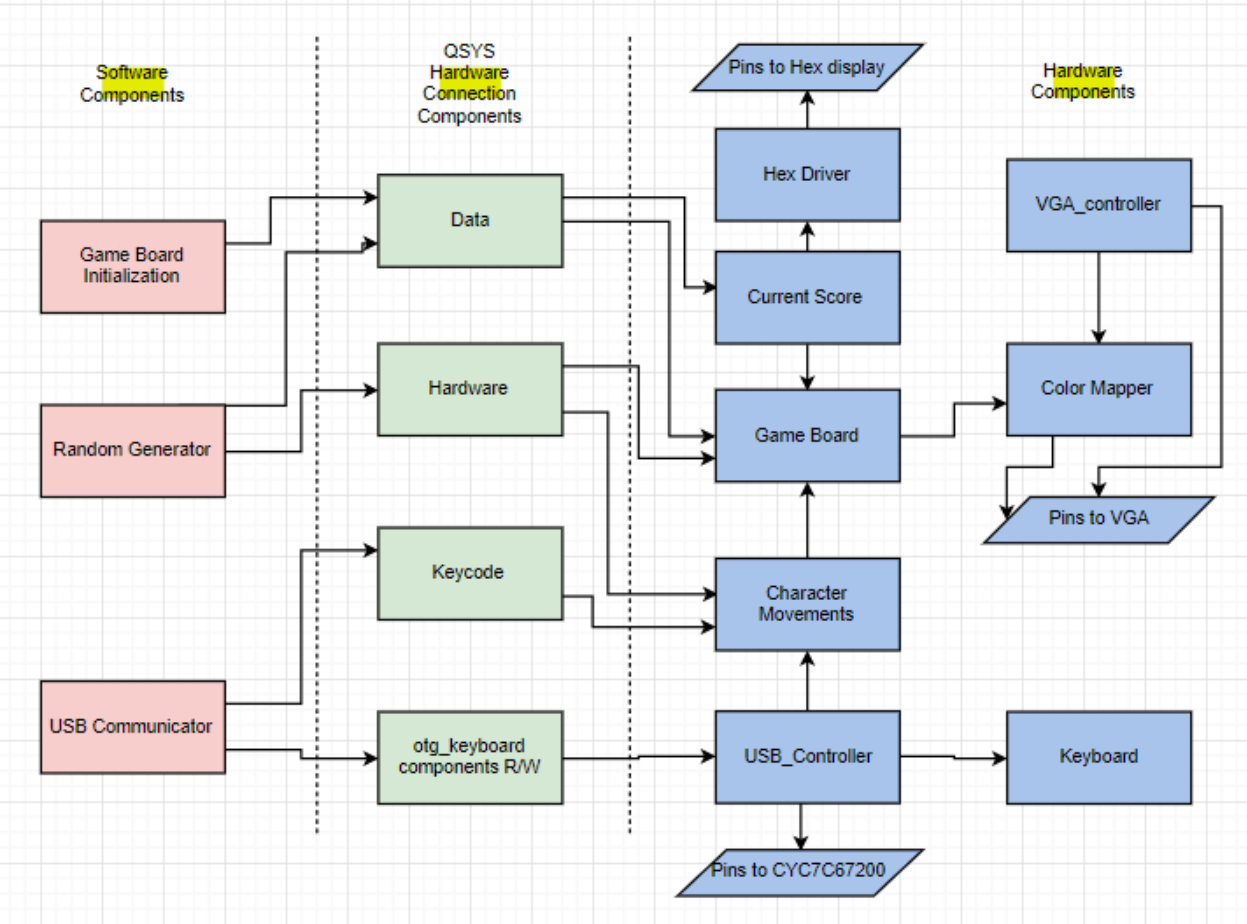


Figure 6: Jodi obtaining a powerup from colliding with Zuofu

Operation of Circuit

Diagram of Circuit Entities



The current game board is initialized with characters generated from software at randomized positions then sent through to the hardware through the data PIO as one big 32-bit data packet with the first 10 bits being the x coordinate, the second 10 being the randomized y coordinate and the next 4 being the randomly generated type of fish. Each fish is generated with different weighted probabilities (i.e.) the starting probabilities has the random generation of the small fish generating at 37%, medium fish at 30%, large fish at 22%, the jellyfish at 7% and the character professor Zuofu generates with a probability of 4%.

In the hardware side, the initialized game board characters take the initialized data packet in the software and loads it into a set of empty registers. We also further impose the maximum fish of each types of fish on the screen at one time using registers specified to a certain type of fish based on the type of the fish that is generated from the fish data that is passed from software side. We also add another random value to make sure that not all fish are load at the same time.

Since we decided to handle the fish position in hardware to increase the efficiency of the program, at every pixel, we used combinational logic that decide what fish sprite will be drawn as well as to handle collision that affect smaller fish and our main character. We utilized the fact that all combinational logic in always_comb is sequential to position our if statements to give use the correct result. Data such as the fish sprite, x position, y position are then passed to color mapper that then choose the SRAM address in which the RGB values of the pixel locates. Next cycle, color mapper then derives the correct RGB value that will be passed to the monitor. The fish will disappear with a clear bit that is set when the character either reaches the end of the screen or if the character is eaten. The registers are reloaded with another packet of random data after they are "killed". The characters are randomly added and are rendered on the screen by a series of sprites which are stored as a HEX file in the SRAM.

The game board information is sent through to the color mapper which map the colors and the VGA controller displays the colors on the screen by loading every corresponding character pixel to the screen. The screen background not containing any characters are set to blue. The screen is then loaded every clock cycle with updated pixel and character locations for their movement.

System Verilog Files

Feeding_frenzy.sv

Inputs:
[3:0] KEY
[1:0] OTG_INT
Outputs:
[6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX6, HEX7
[7:0] VGA_R
[7:0] VGA_G
[7:0] VGA_B
[7:0] VGA_CLK
[7:0] VGA_SYNC_N
[7:0] VGA_BLANK_N
[7:0] VGA_VS
[7:0] VGA_HS
[15:0] OTG_DATA
[1:0] OTG_ADDR
[1:0] OTG_CS_N
[1:0] OTG_RD_N
[1:0] OTG_WR_N
[1:0] OTG_RST_N
[12:0] DRAM_ADDR
[3:0] DRAM_DQM
[3:0] DRAM_RAS_N
[3:0] DRAM_CAS_N
[3:0] DRAM_CKE
[3:0] DRAM_WE_N
[3:0] DRAM_CS_N
[3:0] DRAM_CLK
output(cont.):
[19:0] SRAM_ADDR
[19:0] SRAM_LB_N
[19:0] SRAM_UB_N
[19:0] SRAM_CE_N
[19:0] SRAM_OE_N
[19:0] SRAM_WE_N
Inout:
[31:0] DRAM_DQ
[15:0] SRAM_DQ
Description:
This file is the top-level for the project which contains the initializations for all other modules within the other files Game.sv, Ball.sv, Key_code.sv, Color_Mapper.sv, VGA_Controller.sv, Tristate.sv, Hpi_io_intf.sv, and the Qsys nios_system.qip file generated to create an instance of the nios_system in Quartus.

Game.sv

Inputs:
CLK, VGA_VS, VGA_CLK
RESET
[31:0] fish_data
[15:0] hw
[9:0] Ball_X_Pos
[9:0] Ball_Y_Pos
[7:0] last_key_code
[7:0] key_code_in
[9:0] DrawX, DrawY
Outputs:
[9:0] fish_x
[9:0] fish_y
[3:0] sprite
[15:0] eaten
[3:0] life
Reset_fish
Description:
This file contains all the state logic for the game to be drawn within the Color_mapper. All characters are stored in registers and character actions and movements are declared along with special characteristics that they may have. Also in this file are the states for life and the current score. The logic for the main character's growth is also stored in this file which determines the size of character to be drawn. Game.sv controls the randomized character's movements based off of the DrawX and DrawY pixels on the screen.

Ball.sv

Inputs:
Clk
Reset
frame_clk
[9:0] DrawX, DrawY
[15:0] keycode

Outputs:
is_ball
[9:0] Ball_X_Pos, Ball_Y_Pos
Description:
Ball.sv is almost identical to the Ball.sv file used in lab8 for the bouncing ball. We generate the movement of Jodi there based off of the keycodes obtained from the keypresses of the keyboard. It controls the motions of the main character, Jodi.

Key_code.sv

Inputs:
Frame_clk
[7:0] key_code_in
Outputs:
[7:0] last_key_code
Description:
Key_code.sv is used to store the last key pressed from the keyboard which is then used in game.sv to update the board.

Color_Mapper.sv

Inputs:
Clk, VGA_VS, VGA_CLK, Reset
[15:0] SRAM_out
[9:0] Ball_X_Pos, Ball_Y_Pos
frame_clk
[15:0] hw
[31:0] fish_data
[9:0] DrawX, DrawY
[15:0] keycode
Outputs:
[7:0] VGA_R, VGA_G, VGA_B
[19:0] SRAM_ADDR
[3:0] life
Reset_fish
Description:
Color_Mapper.sv has the logic for drawing the sprites to the screen depending on the locations given to it by the characters stored in the registers in game.sv. The current draw positions are given as DrawX and DrawY. The logic follows the description used in lab8. Draw the pixel by using the sprite arrays stored in the SRAM, if the sprite is within the size boundary for the character within the top left corner + the sprite size, it should draw the sprite with the arrays, otherwise it will draw the background. Since there are 20 different sprites, they are all drawn in a different order according to the layout given in the sprite HEX file. The sprites are created in an array of registers and are then drawn sequentially each time from the list. Each frame must be displayed in one clock cycle for the frame, the data must also be retrieved and drawn at this time as well. The HEX file itself was condensed from 888 color formats where the color is stored in 3, 8bit colors to make 24 bits to 565 color, or 16 bits where the red takes 5 pixels, green take 6 pixels and blue takes 5 pixels. This was done by creating both a Python code to convert to Hex and creating a C code script to shift the bits into 565 color formats.

VGA_Controller.sv

Inputs:
Clk
Reset
VGA_CLK
Outputs:
VGA_HS
VGA_VS
VGA_BLANK_N
VGA_SYNV_N
[9:0] DrawX, DrawY
Description:
We used the VGA_controller.sv from lab 8, the controller controls the time for the VGA through a state machine. It generates the current DrawX and DrawY, or current pixel on the screen which is used to control collisions between the characters, and are created based off the current pixel clock.

Tristate.sv

Inputs:
Clk
Tristate_output_enable
[N-1:0] Data_write
Outputs:
[N-1:0] Data_read
Input:
[N-1:0] Data
Description:
The tristate module is used to ensure that only read and write operations are occurring within the SRAM.

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
Outputs:
[15:0] from_sw_data_in
[1:0] OTG_ADDR
OTG_RD_N
OTG_WR_N

OTG_CS_N
OTG_RST_N
Inout:
[15:0] OTG_DATA
Description:
The hpi_io_intf file chooses the OTG address read, write, chip select and switch data signals coming from the software, it controls the read and write signals coming in, to execute read and write operations in the memory. It is used to read and write from the current keyboard presses which is used in determining Jodi's movement in the ball.sv file.

HexDriver.sv

Inputs:
[3:0] In0
Outputs:
[6:0] Out0
Description:
Inputs 4 bits of a binary input and converts it to 7 bits to be displayed on the Hex values. It controls the LED's on the FPGA board to display the binary values in hexadecimal format.

PIO Block nios_system

Module: nios_system
Inputs:
input wire clk_clk, // clk.clk
input wire [15:0] otg_hpi_data_in_port, // otg_hpi_data.in_port
input wire reset_reset_n, // reset.reset_n
Outputs:
output wire [15:0] keycode_export, // keycode.export
output wire [1:0] otg_hpi_address_export, // otg_hpi_address.export
output wire otg_hpi_cs_export, // otg_hpi_cs.export
output wire [15:0] otg_hpi_data_out_port, // .out_port
output wire otg_hpi_r_export, // otg_hpi_r.export
output wire otg_hpi_w_export, // otg_hpi_w.export
output wire sdram_clk_clk, // sdram_clk.clk
output wire [12:0] sdram_wire_addr, // sdram_wire.addr
output wire [1:0] sdram_wire_ba, // .ba
output wire sdram_wire_cas_n, // .cas_n
output wire sdram_wire_cke, // .cke
output wire sdram_wire_cs_n, // .cs_n
output wire [3:0] sdram_wire_dqm, // .dqm
output wire sdram_wire_ras_n, // .ras_n
output wire sdram_wire_we_n // .we_n
Inouts
inout wire [31:0] sdram_wire_dq, // .dq
Description: This module takes in data, clock and reset signal and wire all the signals to the correct submodules
Purpose: This module is the top-level module that takes in inputs and connect all the wire to the right submodule to produce the correct result

Module: nios_system_otg_hpi_data

Inputs:
input [1: 0] address;
input chipselect;
input clk;
input [15: 0] in_port;
input reset_n;
input write_n;
input [31: 0] writedata;

Outputs:
output [15: 0] out_port;
output [31: 0] readdata;
Description: This module assign data out as 0 if reset, else to write_data. This module also assigns readdata the correct value from the mux (the right register), and assign the out_port value to the data out
Purpose: This module is used to input or output data

Module: nios_system_otg_hpi_cs

Inputs:
input [1: 0] address;
input chipselect;
input clk;
input reset_n;
input write_n;
input [31: 0] writedata;
Outputs:
output out_port;
output [31: 0] readdata;
Description: This module assign data out as 0 if reset, else to write_data. This module also assigns readdata the correct value from the mux (the right register), and assign the out_port value to the data out
Purpose: This module is used to determine the output chip select

Module: nios_system_otg_hpi_address

Inputs:
input [1: 0] address;
input chipselect;
input clk;
input reset_n;
input write_n;
input [31: 0] writedata;
Outputs:
output [1: 0] out_port;
output [31: 0] readdata;
Description: This module assign data out as 0 if reset, else to write_data. This module also assigns readdata the correct value from the mux (the right register), and assign the out_port value to the data out
Purpose: This module is used to determine the output address

Module: nios_system_jtag_uart_0

Inputs:
input av_address;
input av_chipselect;
input av_read_n;
input av_write_n;
input [31: 0] av_writedata;
input clk;
input rst_n;
Outputs:
output av_irq;
output [31: 0] av_readdata;
output av_waitrequest;
output dataavailable;
output readyfordata;
Description: This module takes in different signals and output the correct read data and signal that show it is ready for data as well as the wait request signal.
Purpose: This module controls the flow of data

Module: nios_system_keycode

Inputs:
input [1: 0] address;
input chipselect;
input clk;
input reset_n;
input write_n;
input [31: 0] writedata;
Outputs:
output [15: 0] out_port;
output [31: 0] readdata;
Description: This module assign data out as 0 if reset, else to write_data. This module also assigns readdata the correct value from the mux (the right register), and assign the out_port value to the data out
Purpose: This module use take output the keycode values

Software Files

main.c

Main.c utilizes the main.c from lab8 for usb inputs along with our modification of a game state. It is where the board game logic gets initialized. Each fish has its own respective position contained in the board generated from a random generator which is also implemented as a function in the main. The random generator generates random numbers based off weighted probabilities given by the user in the code to generate a weighted random set of small, medium, and large characters.

usb.c/usb.h

These files were used from lab8 to get the keyboard input data. It has methods used for our keyboard usb interface.

Figure 7: Jodi's upward movement

Sprites

Each sprite is assigned a register in game.sv and contains its own logic of its position on the screen. Each fish is also given a random speed between 1-6 to move across the board. Two of the sprites, Zuofu are not killable and cause unfortunate progress delays in the main character, Jodi's, overall score and progress within the game. We created our own sprites for the game because we did not like the design of any characters that we saw online.

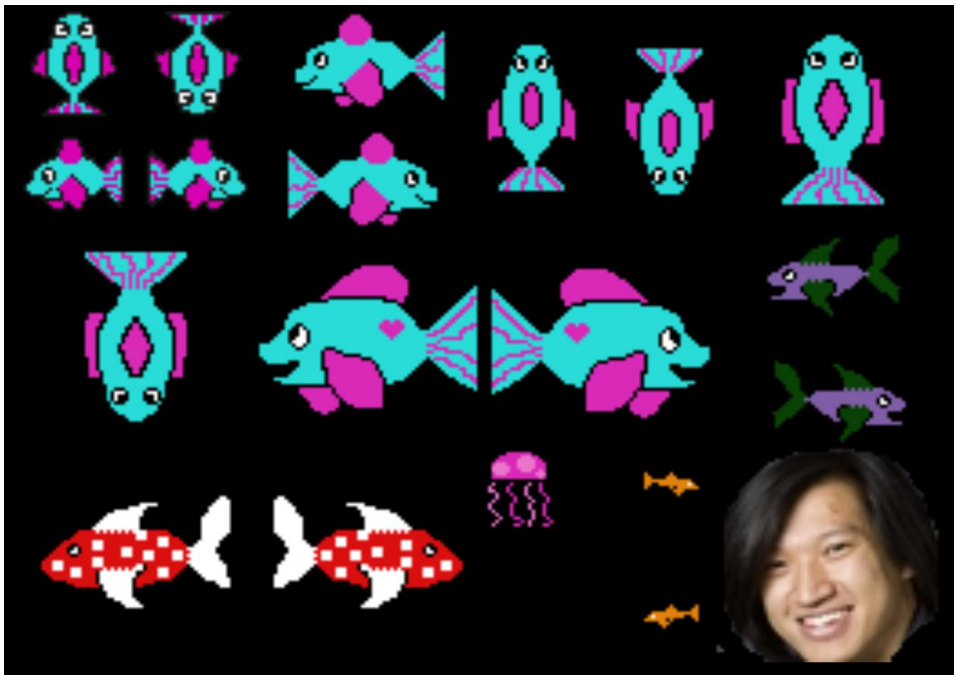


Figure 8: created sprite table

Character - Jodi:

Jodi is the main character of the game and has three sizes small, medium, and large. The user controls Jodi's movements through the keyboard and depending on the amount of fish consumed/eaten.

Character - Medium fish:

The medium fish is randomly generated on the game screen and given a random speed, it also can eat small sized fish and small sized Jodi. It can be eaten by medium sized Jodi. The medium fish has a spawn rate of 30%

Character - Large fish:

The large fish is randomly generated on the game screen and is given a random speed, it also can eat medium sized Jodi and the small and medium sized fish and can be eaten by large sized Jodi. The large fish is given a 22% chance of spawning.

Character - Small fish:

The small fish is randomly generated on the game screen and is given a random speed, it cannot eat any other fish and can be eaten by the small sized Jodi and can be eaten by any of the larger fish. Small fish are given a 37% chance of spawning onto the game board.

Character - Zuofu:

Zuofu is a bonus in the game. If Jodi consumes Zuofu Jodi gains a 10-point bonus, which enables Jodi to increase a size, because Jodi's size increases every time 10 fish are consumed. Zuofu has a 4% chance of spawning in the game world.

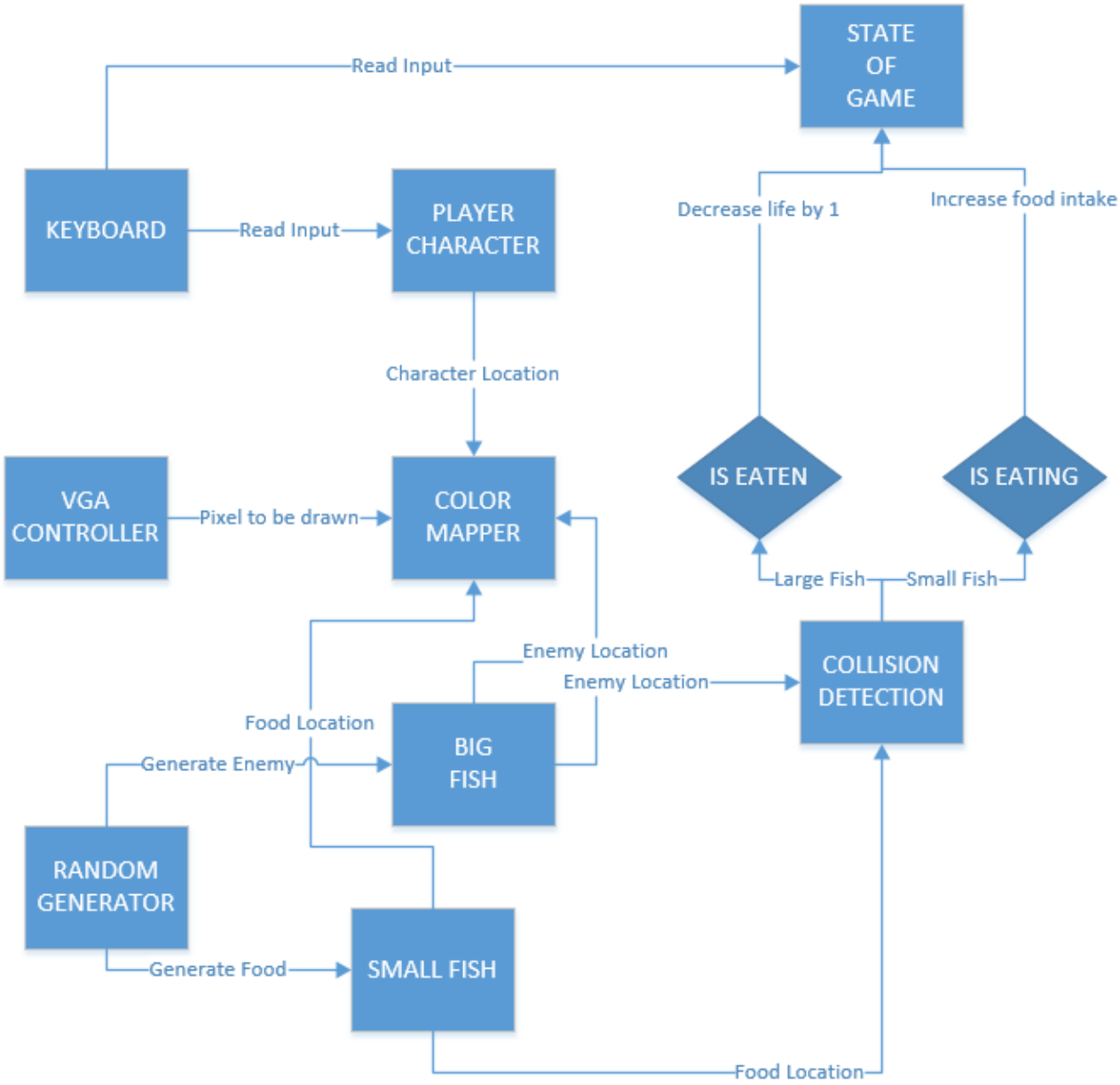
Character - Jellyfish:

The jellyfish is an enemy in the game and every time Jodi touches the jellyfish, Jodi loses 10 points on the overall score, making Jodi shrink down a size. The jellyfish has a 7% chance of spawning within the game world.

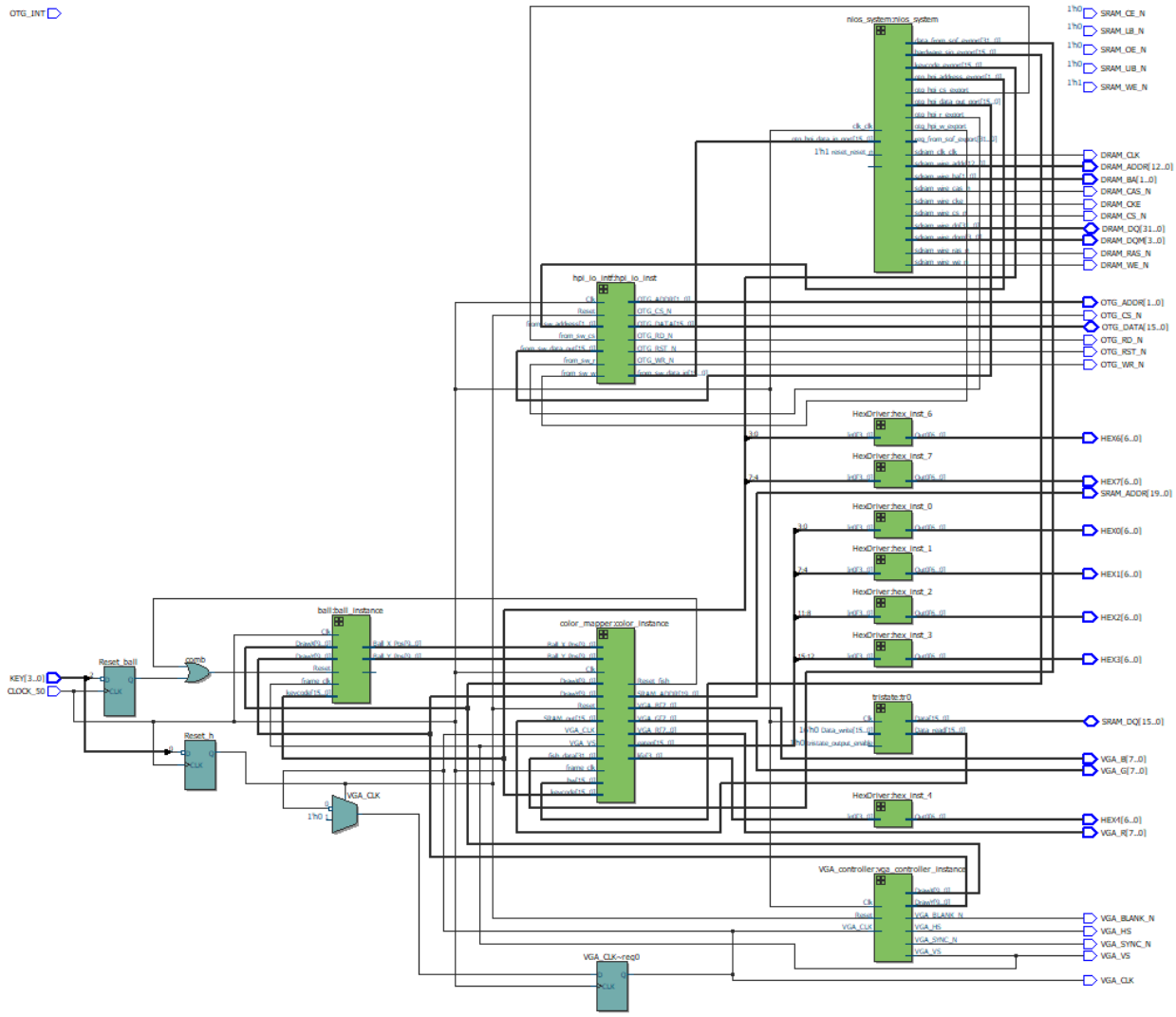
Block Diagram

The block diagram is designed to show the high-level design for the diagram there will be a function to read from the keyboard, generate random characters (food/enemies) a color mapper to refresh the current on-screen graphics, and a module to detect collisions to determine whether the character should become larger from consuming food or decrease the number of lives if it has collided with an enemy.

Block Diagram high level



Block Diagram Components



Conclusion

In the end, we were able to meet and exceed our baseline set of features that we had originally set for our game, the baseline set was simply to have a character consume and be consumed by other characters, have a set amount of lives, the ability to increase to consume large characters, and a display for the current number of lives and current score depending on the number of enemies consumed. Originally, the goal was to make the characters circles that could increase in size and consume other circles. Instead we created and designed our own sprites and loaded them onto the SRAM to use within the game. We also created a powerup in the form of Zuofu, and an enemy in the form of a jellyfish. The final add-on we included was a title page that requires the player to press the spacebar to start the game. Jodi was also set to respawn in the middle of the game with no other fish around her when the player died. Overall, this was a very challenging project that was a lot of fun to make and follow through to the end.

A few things we would have liked to include in the game if we had extra time would be the addition of an undersea background for Jodi, we ended up using a plain blue screen, but we would have liked to use the picture that we ended up including in the title sequence of the game. We would also like to change the probability of the types of fish that spawn over time to make the game even more challenging as the player continues to increase in size.

Overall, our project was fun to make and through it, we learned a lot more about System Verilog. One difficulty we had was getting the characters to load from the SRAM and having them appear on screen using the random generator we created. Originally, we had planned on using the on-chip memory to load the characters, but after trying to load several characters at once, on-chip did not have enough storage, so we had to switch to SRAM instead. We had to create both a python script to convert our array of colors in C to a hex file to upload it into the SRAM. Another obstacle which came from this was the fact that the colors displayed in the Hex value were red as BGR instead of RGB by the SRAM, so we had to implement a method to swap the colors within System Verilog as well. Another difficulty that we came across was the long compile times and buggy crashes within the eclipse software. We managed to both figure out how to fix many errors that would display in the eclipse software along with decrease our overall runtime in Quartus, both of which allowed us to increase our overall productivity. Because of all the work we put in this project, it has enabled us to be prepared for future FPGA projects and designs.