ECE 385

Fall 2021

Final Project

I Wanna Be the Guy

Guanshujie Fu 3190110666

Xiaomin Qiu 3190110717

Lab Section D231 18:00 – 21:00

TA: Yuchuan Zhu

1. Introduction

We have implemented a game inspired by ‘I wanna’ series which is famous for its difficulty as our final project. Our version is a bit different and a bit simpler, but still just as fun. Using a keyboard, a player can control a kid to move forward, back and jump twice to avoids all kinds of traps to reach the destination.

The game uses SDRAM to store the image sprites, SRAM as double buffer, on-chip memory to store the NIOS C program, NIOS II SOC to support keyboard control and VGA to display the game. The realization of the VGA display for background and characters depends on the structure and storage of the corresponding images, as well as our pixel access mechanism. We have grouped character images that correspond to all frames of different actions for each character into one sprite sheet. And we have chosen to store the background image in SRAM and kid’s sprite sheets in on-chip memory. Our implementation of kid’s movements is mainly based on the algorithm for ball movement in lab8. We inherited the code for both USB keyboard interface and VGA interface. The implementation, other than interfacing with the keyboard, which is done using the NIOS II CPU, is done in hardware.

2. Written Description and Diagrams

i. ***<u>General Descreption</u>***

As has been implemented in lab 8, the NIOS system fetch user’s inputs from CY7 USB chip, and then transfer the signal to ball routing and VGA controller to control the VGA monitor to display the corresponding position of the ball. The USB keyboard needs to communicate with the NIOS II through CY7 chip and HPI as the interface. The VGA monitor receives signals generated in the FPGA and use them to display each pixel on the monitor. The color mapper keeps drawing images to VGA. With the signals DrawX, DrawY telling which pixel VGA is currently drawing, color mapper decides which color (red, green, and blue) to map into the corresponding pixel. The module kid.sv tells the status of the character. It’s connected to keyboard by NIOS II system, which outputs keyboard values to the kid. Given DrawX and DrawY, the current drawing position of the grounds, the modules would decide whether their corresponding objects (grass ground, non-grass ground, masked ground) are at that location, and send out a signal. Color mapper decides which object to draw with some priority. In our project, pictures are stored in On-chip Memory in the form of palette indices, which is demonstrated on ECE385 Helper Tools. To save On-chip Memory as well as to simplify address calculation, we use separate sprites for different objects. With the helper tools, we draw a lot of blocks in our game.

We also add animation to the kid to make our game look more fun. As the figure shown, the kid has four major motions and each of the motion is constructed with some different images. We could use them to make an animation for the kid, to achieve this, we need some logic tell which status the kid is in, either walking, jumping, falling or idle, and use different sprites for different status. To add animation to each status, we just need to alter the sprites used to draw the kid. Circularly switching among the set of images in the sprite results in animation. To simplify calculation, we make the images arrange closely in one row, each image having the same size. Then, an index selecting which image to draw in the current cycle is needed when calculating the drawing address. We call the index Walking_Counter, since it adds 1 repetitively, and clears to zero when maximum is reached. We still need something to trigger that addition, which determines how fast the animation runs. Thus, we can create a clock triggering animation.

ii. Overview of the design procedure

a. What code is used as the foundation of the project?

The final.sv is used as the top-level of the project, which is hence the foundation of the project.

b. What are the different objectives of the project?

For this project, multiple objectives are implemented. For instance, several state machines are implemented for different parts of the project to deal with the moving logics of the kid and the game logic. Also, sprites are used to implement the images of background, kids and other elements in the game. Moreover, we utilize the On-Chip Memory to store the images of all elements in the game.

c. What research/background study has been done to achieve the objectives?

Before the start of the project, several research are done to achieve the objectives mentioned above. We have learnt how to use the provided tools to convert the images from png into other storable forms like txt and how to use On-Chip Memory to store the converted images. We also check the storing capacity of the On-Chip Memory to choose images of appropriate sizes.

d. <u>How are the different objectives linked together to form a complete project?</u>

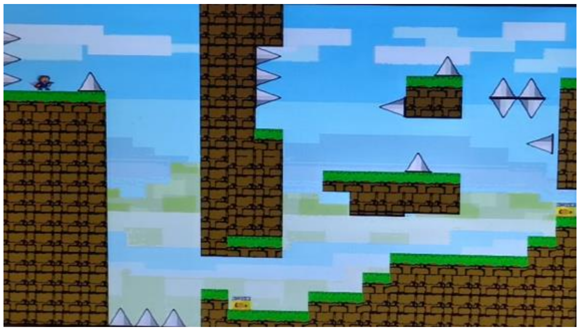
Different objectives work together and compose the whole logic of the game. For instance, the state machines we implemented are closely linked to the modules of the kids and grounds, which is then linked to the use of sprites and memory.

iii. ***<u>Start Interface</u>***

This component is the first component of the game. It has a cool background image. The game is stuck in this component until 'space' is pressed.

iv. ***<u>Game Interface</u>***

The drawing space of our game is 640*480, and the size of a typical ground block, kid and stab is 30 * 30 pixels. We arrange the blocks one by one as the logic used to arrange combinational blocks would be more complex and would cause more bugs that need to find. Arrange the pixels one by one also makes the map more flexible.



v. ***<u>Collision Detection</u>***

Collision detection is the most important part of any fighting game. Without collision detection, the kid cannot properly interface with the environment. The difficulty of collision detection lies in the number of situations in which a collision occurs. There are 4 situations that count as one collision, each depending not only on the kid's position, but also on their direction. In our game, it is implemented by a rectangular box approach. In this method, the kid is represented by a rectangular box, which is considered as its impact box. If this rectangle overlaps with any obstacle, the game over flag is set to high. Most of the collision detection is implemented on the obstacle itself. The kid's X, Y and size are passed to all obstacles, so we can check in the obstacles module if the kid's rectangular hit box has hit an obstacle. However, in kid.sv itself, we also implement collision detection for ceiling and floor by checking the y position of the kid. If this y-position is greater than the maximum height or less than 0, the game over flag is also set. In the obstacle module, we derive an algorithm to check if some part of the kid collides with an obstacle. First, the top of the child is checked to see if it has hit an obstacle. After that, we use formulas to check if the bottom of the child collides with an obstacle. Essentially, kid_x and kid_y record the leftmost and topmost positions of the sprite. Therefore, we check if the bottom of the kid is larger than the topmost position of the obstacle and smaller than the bottommost position of the obstacle.

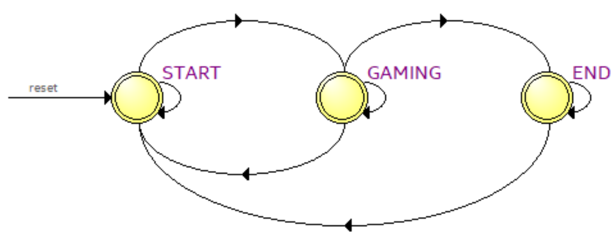
vi. <u>Boundaries</u>

In the early development of our game, we had opted for a hard wall boundary, meaning the left and right side of the screen was the end of the game screen and you cannot go through. However, we experienced a weird bug in that the players were able to go through the left side of the screen. Thus, we added thorn at the left side and implement it as the trap.

vii. <u>Block diagram / State Diagram</u>

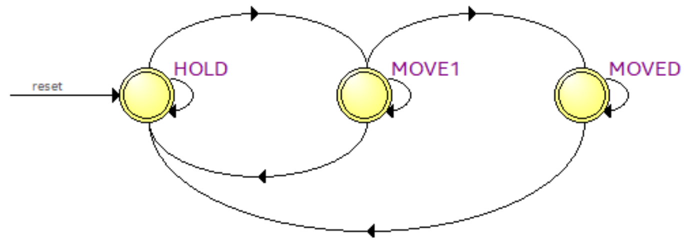
a. <u>State Machine Diagram for the game</u>

All state machines we designed are Mealy Machine as they all depend on the input from different part of the project like the keyboard operation or current position of the kid.



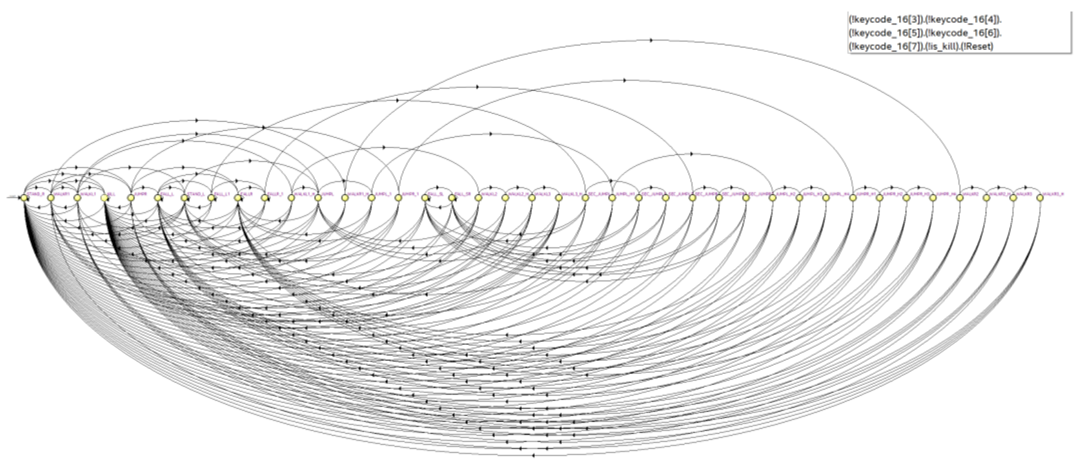
As you can see from the diagram, the logic for the whole game is simple and direct, which is consisted of three states START, GAMING and END. At the start page of the game, it is at START state. When space is pressed and the game starts, the game comes into GAMING. When you win or die, the game comes into END.

b. <u>State Machine Diagram for the moving elements in the game</u>



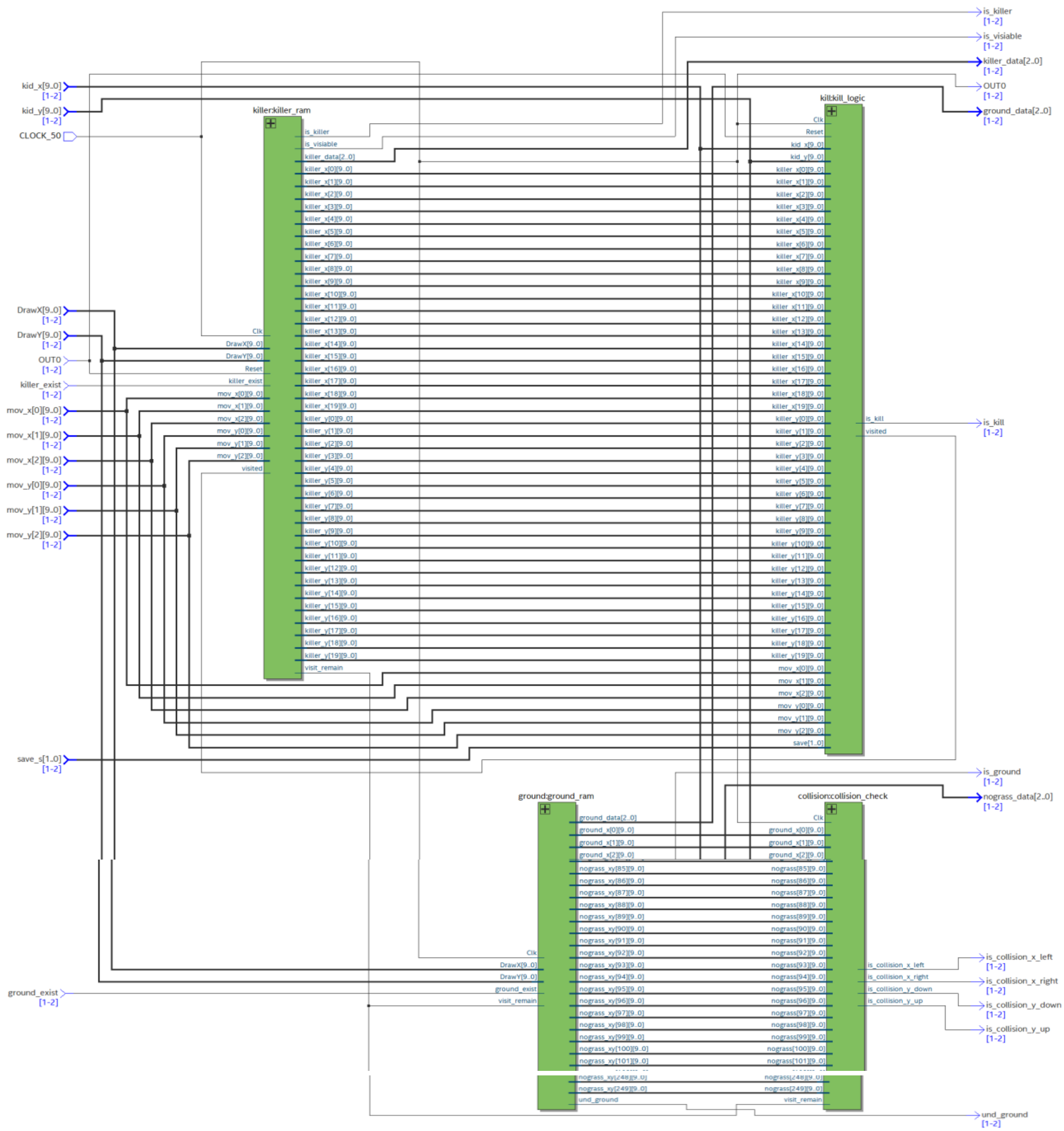
This state machine is to deal with some operation sensitive elements. For instance, some traps are linked to the position of the kid and the movement of the kid. When the condition is not satisfied, the state of the trap is HOLD and when it is triggered, the state will shift to MOVE1. Once moved, the state will remain in MOVED until the game is reset or restart.

c. *State Machine Diagram for the moving logic of kid*



As you can see, the state machine for this part of logic is extremely complex and hence only a small part of states are shown here. We use 35 states to deal with the movement of the kid like Walk_right, Walk_left or Jump_right .etc. Depending on the operation of the player, state will shift and each will generate a 32-bit signal indicating the motion of the kid. The 32-bit signal will be used by modules that draw the kid on the screen, which choose different images of the kid accordingly.

d. *Block Diagram*



3. Module descriptions

Modules	**2_killer_FSM.sv**
Inputs	input logic Clk, input logic Clk_s, input logic Reset, input logic restart, input logic [9:0] kid_x, kid_y

Outputs	output logic [2:0][9:0] mov_x, mov_y, output logic [2:0] states, output logic move
Description	State machine for the moving stab. It has three states for each moving stab, including HOLD, MOVE1, MOVED state.
Purpose	The moving stabs construct a challenging trap for the game. The kid need to find and then dodge the trap to survive.

Modules	**kill.sv**
Inputs	input logic Clk, input logic Reset, input logic [19:0][9:0] killer_x, input logic [19:0][9:0] killer_y, input logic [2:0][9:0] mov_x, mov_y, input logic [9:0] kid_x, kid_y, input logic [1:0] save,
Outputs	output logic visited, output logic is_kill
Description	It uses multiplexer to judge whether the kid touch the stab and should the kid be killed based on the distance between them.
Purpose	It forms the most important nature of the trap, which would kill the kid when he touch it.

Modules	**Killer_ram.sv**
Inputs	input Clk, input Reset, input logic killer_exist, input logic visited, input logic [9:0] DrawX, DrawY, input logic [2:0][9:0] mov_x, mov_y
Outputs	output logic [19:0][9:0] killer_x output logic [19:0][9:0] killer_y, output logic [2:0] killer_data, output logic is_visiable, output logic visit_remain,

	output logic is_killer
Description	Record the coordinates of the stabs, trigger mechanism and movement mode of the traps.
Purpose	It defines the location and orientation of the traps and the type of the traps, such as visible, non-visible, moving.

Modules	**collision.sv**
Inputs	input logic Clk, input logic [28:0][9:0] ground_x, input logic [28:0][9:0] ground_y, input logic [249:0][9:0] nograss, input logic [9:0] kid_x, kid_y, input logic visit_remain,
Outputs	output logic [2:0] counter_right, counter_left, counter_up, counter_down output logic is_collision_x_right, is_collision_x_left, output logic is_collision_y_down, is_collision_y_up
Description	It uses multiplexers to judge the collisions between kid and grass grounds, non-grass grounds and masked grounds.
Purpose	It is used to avoid collision between kid and the grounds through a rectangular box method. In this method, the kid was represented by a rectangular box that was considered its hit box. If this rectangle overlapped with any of the obstacles, the game over flag was set high.

Modules	**0_FSM.sv**
Inputs	input logic Clk, input logic Reset, input [7:0] keycode_16 input logic is_kill,
Outputs	output logic start_r, output logic startpage_exist, output logic background_exist output logic kid_exist, output logic ground_exist, output logic killer_exist, output logic save_exist, output logic endpage_exist

Description	Finate state machine for the whole game the States for the whole game is: Start -> Gaming -> End At Start state, only start page shows. At Gaming state, the game logic should be presented. At End state, come to end page and return to Start page.
Purpose	If kid is killed, game end game starts or restarts, it works similar to Reset.

Modules	**1_kid_FSM.sv**
Inputs	input logic Clk, input logic restart, input logic Reset, input logic is_kill, input logic is_collision_y_down input logic [15:0] keycode_16,
Outputs	output logic [15:0] kid_state
Description	State machine of the kid, used to perform the animation of kid’s movement, including fall, stand, walk and jump. Each movement has some different states.
Purpose	35 states are utilized to implements the motions of the kid, which is shifted according to the inputs from keyboard and current states.

Modules	**Ground_ram.sv**
Inputs	input Clk, input logic ground_exist, input logic visit_remain, input logic [9:0] DrawX, DrawY
Outputs	output logic [28:0][9:0] ground_x output logic [28:0][9:0] ground_y output logic [249:0][9:0] nograss_xy output logic [2:0] ground_data output logic [2:0] nograss_data output logic is_ground output logic und_ground
Description	Record the coordinated of all the grass ground, non-grass grounds and masked grounds. It also records the trigger mechanism of the masked ground.

Purpose	This module stores the image of ground into On-Chip Memory and output corresponding data of the ground into Color Mapper to draw the background on the screen

Modules	**kid.sv**
Inputs	input Clk, Reset, frame_clk, input logic is_collision_x_right, is_collision_x_left input logic is_collision_y_down, is_collision_y_up, input logic is_kill, input logic restart, input logic [7:0] kid_jump_states input [9:0] DrawX, DrawY input [15:0] keycode_16
Outputs	output logic [9:0] kid_x, kid_y output logic [9:0] counter_out
Description	It is used to control the movement of the kid in X and Y directions by changing the pixels the ball located, depending on the keyboard inputs. Through VGA output, the corresponding output will be displayed on the screen.
Purpose	Used to control the kid's motion and position. It makes the kid move on the VGA monitor screen. The entire system contains three main parts: NIOS II processor, USB keyboard, and VGA monitor. The NIOS system fetch user's inputs from CY7 USB chip, and then transfer the signal to kid routing and VGA controller to control the VGA monitor to display the corresponding position of the ball. The USB keyboard needs to communicate with the NIOS II through CY7 chip and HPI as the interface. The VGA monitor receives signals generated in the FPGA and use them to display each pixel on the monitor.

Modules	**Save_ram.sv**
Inputs	input Clk, input logic save_exist, input logic [9:0] DrawX, DrawY input logic [9:0] kid_x, kid_y input logic [7:0] keycode,
Outputs	output logic [2:0] save_data

	output logic is_save, output logic [1:0] save_s
Description	
Purpose	This module stores the image of saver into On-Chip Memory and output corresponding data of the saver into Color Mapper to draw the background on the screen

Modules	**Kid_ram.sv**
Inputs	input Clk, input logic kid_exist, input logic [15:0] kid_state, input logic is_kill, input logic [9:0] DrawX, DrawY input logic [9:0] kid_x, input logic [9:0] kid_y,
Outputs	output logic [3:0] kid_data output logic is_kid
Description	With the state and coordinates of the kid when moving right/left and the current pixel coordinates, this module could calculate the index of kid's background color to shown kid on the screen.
Purpose	This module mainly implements two functions as it stores all images of different motions of kids into the On-Chip Memory and output corresponding data of the kids into Color Mapper to draw the kid on screen.

Modules	**Background_ram.sv**
Inputs	input Clk, input logic background_exist, input logic [9:0] DrawX, DrawY
Outputs	output logic [3:0] background_data output logic is_background
Description	Current pixel coordinates To index of background color
Purpose	This module stores the image of background into On-Chip Memory and output corresponding data of the background into Color Mapper to draw the background on the screen.

Modules	hpi_io_intf.sv
Inputs	input Clk, Reset, input [1:0] from_sw_address input from_sw_r, from_sw_w, from_sw_cs, from_sw_reset input [15:0] from_sw_data_out
Outputs	output[15:0] from_sw_data_in inout [15:0] OTG_DATA, output[1:0] OTG_ADDR, output OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N
Description	This module is used to interfaces with OTG chip which controls the USB and determines the outputted signals. The otg-variables are the inputs of the USB chip, and they are updated to the sw-variables as required.
Purpose	This module controls and updates otg signals like read, write, address, databuffer signals to the OTG chip which ensure the correct data reads, writes and reset from USB keyboard for C#. It allows us to control the input output bus.

Modules	VGA_controller.sv
Inputs	input Clk, Reset, input VGA_CLK,
Outputs	output logic VGA_HS, VGA_VS, output logic VGA_BLANK_N, VGA_SYNC_N, output logic [9:0] DrawX, DrawY
Description	This module controls the VGA signals such as the vertical and horizontal syncs. It takes the data of the ball's current position on screen and updates it to the VGA display with the variables DrawX and DrawY.
Purpose	This module is used to determine the edge of the screen and when to output the vertical or horizontal sync signals. It also records the position of the pixel we are working on.

Modules	Lab8.sv
Inputs	input CLOCK_50, input[3:0] KEY,

	input OTG_INT,
Outputs	output logic [6:0] HEX0, HEX1, output logic [7:0] VGA_R, VGA_G, VGA_B, output logic VGA_CLK, VGA_SYNC_N, output logic VGA_BLANK_N, VGA_VS, VGA_HS, inout wire [15:0] OTG_DATA, output logic[1:0] OTG_ADDR, output logic OTG_CS_N, OTG_RD_N, output logic OTG_WR_N, OTG_RST_N,
Description	It is the top-level module which connects NIOS 2, OTG chip connection and the VGA controller modules.
Purpose	It links all the modules together and instantiates all the System Verilog modules.

Modules	**HexDriver.sv**
Inputs	input logic [3:0] In0
Outputs	output logic [6:0] Out0
Description	This is a hex driver that transfers a 4-bit signal to represent a hex display on the FPGA. The 4 bits in are needed to have designations for hex values 0 to F.
Purpose	It helps to present the output hex-value on the board.

Modules	**color_mapper.sv**
Inputs	input is_ball, input[9:0] DrawX, DrawY,
Outputs	output logic [7:0] VGA_R, VGA_G, VGA_B
Description	This module determines the color that should be displayed on the monitor.
Purpose	It is used to decide which color to be output to VGA for each pixel. The input of this module determines if the current pixel is a kid. The pixel will become white if is_kid, and stay in the background color if! is_kid.

Connections	Name	Description	Export	Clock	Base	End	...	Tags	Opcode Name
	clk_0	Clock Source							
	clk_in	Clock Input	clk	export					
	clk_in_reset	Reset Input	reset						
	clk	Clock Output	Double-clic	clk_0					
	clk_reset	Reset Output	Double-clic						
	nios2_gen2_0	Nios II Proce...							
	clk	Clock Input	Double-clic	clk_0					
	reset	Reset Input	Double-clic	[clk]					
	data_master	Avalon Memory...	Double-clic	[clk]					
	instruction_master	Avalon Memory...	Double-clic	[clk]					
	irq	Interrupt Rec...	Double-clic	[clk]	IRQ 0	IRQ 31			
	debug_reset_request	Reset Output	Double-clic	[clk]					
	debug_mem_slave	Avalon Memory...	Double-clic	[clk]	0x1000	0x17ff			
	custom_instructi...	Custom Instru...	Double-clic						
	onchip_memory2_0	On-Chip Memor...							
	clk1	Clock Input	Double-clic	clk_0					
	s1	Avalon Memory...	Double-clic	[clk1]	0x0	0xf			
	reset1	Reset Input	Double-clic	[clk1]					
	sdram	SDRAM Control...							
	clk	Clock Input	Double-clic	sd...					
	reset	Reset Input	Double-clic	[clk]					
	s1	Avalon Memory...	Double-clic	[clk]	1000_0000	17ff_ffff			
	wire	Conduit	sdram_wire						
	sdram_pll	ALTPLL Intel ...							
	inclk_interface	Clock Input	Double-clic	clk_0					
	inclk_interface_...	Reset Input	Double-clic	[in...					
	pll_slave	Avalon Memory...	Double-clic	[in...	0xa0	0xaf			
	c0	Clock Output	Double-clic	sdr...					
	c1	Clock Output	sdram_clk	sdr...					
	sysid_qsys_0	System ID Per...							
	clk	Clock Input	Double-clic	clk_0					
	reset	Reset Input	Double-clic	[clk]					
	control_slave	Avalon Memory...	Double-clic	[clk]	0xb8	0xbf			
	keycode	PIO (Parallel...							
	clk	Clock Input	Double-clic	clk_0					
	reset	Reset Input	Double-clic	[clk]					
	s1	Avalon Memory...	Double-clic	[clk]	0x90	0x9f			
	external_connection	Conduit	keycode						
	otg_hpi_address	PIO (Parallel...							
	clk	Clock Input	Double-clic	clk_0					
	reset	Reset Input	Double-clic	[clk]					
	s1	Avalon Memory...	Double-clic	[clk]	0x80	0x8f			
	external_connection	Conduit	otg_hpi_...						
	otg_hpi_data	PIO (Parallel...							
	clk	Clock Input	Double-clic	clk_0					
	reset	Reset Input	Double-clic	[clk]					
	s1	Avalon Memory...	Double-clic	[clk]	0x70	0x7f			
	external_connection	Conduit	otg_hpi_...						
	otg_hpi_r	PIO (Parallel...							
	clk	Clock Input	Double-clic	clk_0					
	reset	Reset Input	Double-clic	[clk]					
	s1	Avalon Memory...	Double-clic	[clk]	0x60	0x6f			
	external_connection	Conduit	otg_hpi_r						
	otg_hpi_w	PIO (Parallel...							
	clk	Clock Input	Double-clic	clk_0					
	reset	Reset Input	Double-clic	[clk]					
	s1	Avalon Memory...	Double-clic	[clk]	0x50	0x5f			
	external_connection	Conduit	otg_hpi_w						
	otg_hpi_cs	PIO (Parallel...							
	clk	Clock Input	Double-clic	clk_0					
	reset	Reset Input	Double-clic	[clk]					
	s1	Avalon Memory...	Double-clic	[clk]	0x40	0x4f			
	external_connection	Conduit	otg_hpi_cs						
	otg_hpi_reset	PIO (Parallel...							
	clk	Clock Input	Double-clic	clk_0					
	reset	Reset Input	Double-clic	[clk]					
	s1	Avalon Memory...	Double-clic	[clk]	0x30	0x3f			
	external_connection	Conduit	otg_hpi_...						
	jtag_uart_0	JTAG UART Int...							
	clk	Clock Input	Double-clic	clk_0					
	reset	Reset Input	Double-clic	[clk]					
	avalon_jtag_slave	Avalon Memory...	Double-clic	[clk]	0xc0	0xc7			
	irq	Interrupt Sender	Double-clic	[clk]			5		
	keycode2	PIO (Parallel...							
	clk	Clock Input	Double-clic	clk_0					
	reset	Reset Input	Double-clic	[clk]					
	s1	Avalon Memory...	Double-clic	[clk]	0x20	0x2f			
	external_connection	Conduit	keycode2						

Clk_0: It serves as the functional clock for the entire system and as a reference for other modules.

nios2_gen2_0: It handles the conversion of the C code into system verilog, which can then be executed on the hardware FPGA board.

Sdram: It allows us to get available sdram on the FPGA board, since the on-chip memory is too small to successfully store and update the available programs. sdram is very useful due to its fast-processing time and low update and output latency.

sdram_pll: This block is a way to account for the small latency within the transfer of data to and from the SDRAM and serves as a separate clock for the system.

sysid_qsys_0: This block verifies correct software and hardware transfers by looking back and forth between C code and SystemVerilog to ensure that the data is transferred in the correct format.

jtag_uart_0: It allows terminal access and is used to debug the software.

otg_hpi_r: It is the PIO that makes the enable bit to read from the SOC memory which is sent from the software to the FPGA.

otg_hpi_w: It is the PIO that makes the enable bit to write to the memory of the SoC that is sent from the software to the FPGA.

otg_hpi_cs: It is the PIO that makes the enable bit to turn on and off the memory of the SOC that is sent from the software to the FPGA.

otg_hpi_reset: It is the PIO that makes the reset of the SOC memory sent from the software to the FPGA possible.

4. List of Features

i. *Fill out the design resources and statistics table (duplicated here for convenience).*

LUT	7593
DSP	30
Memory (BRAM)	704692
Flip-Flop	2373
Frequency	17.92MHz
Static Power	105.54mW
Dynamic Power	0.85mW
Total Power	192.22mW

ii.

5. Timeline Predictions and Results

i. *Week1(11/9-12/2):proposal*

We have discussed the feasibility of several plans and we conducted an analysis of the element requirements of the project realization to make a proposal. We decide to choose 'I wanna be the guy' as our final project as it is realizable and could apply most of the knowledge we learnt this semester.

ii. *Week2(12/2-12/9):*

We modified the existing Lab 8 movement code to allow jump twice according to active key presses. We succeeded at loading 3 Link sprites, facing up, down, and to the right/left. Also, we successfully design the start interface and the testing game interface.

iii. *Week3(12/9-12/16):midchek*

This week, we focused on state transition logic and debugging sprite loading. This week was mainly dedicated to creating the ground module, which took significant time and debugging, as we struggled to deal with the collision problem. We also create the killer module, the stabs, which will be used to create traps. Making the kid move on the ground was easy, but creating die transition logic which should be triggered by touching the trap was more challenging than we thought.

iv. *Week4(12/16-12/23):*

We began work on the animation of the objects, as well as debugged kid movement to meet our design constraints. We had issues with the enemy turning around too fast, and it took us some time and unhelpful advice before we ended up at the desired speed. It took us some time to deal with the images of the kid's motion, which will be used to construct the animation. We need to animate the kid in the desired direction and location (in front of Link depending on the direction he was facing).

v. *Week5(12/23-1/4):demo*

These two weeks we focused on getting ready for our demo. We debugged any sprite issues, optimized the loading to improve compile time, and started on the final game interface designing and testing. We need to draw the game map using the modules we designed, and make it interact with the kid to make the game look more fun and challenging.

Fgsj==

6. Conclusion

In conclusion, the design can successfully perform the game and all parts are functioning properly. We spend a lot of time writing this project, which offers us a chance to get in touch with the development of a game. We enjoy the process of creating and developing something from empty. We also have a deeper understanding of how hardware and software work together, which might have unexpected benefit for our future development.

7. Codes

The source of the codes for our project is open and has been attached with this report and also uploaded to GitHub.