ECE 385 Lab Report

Spring 2023

Final Project: PokeHead Video Game on FPGA

Name: Zhu Hanggang, Hong Jiadong

Student ID: 3200110457, 3200110970

Prof. Chushan Li, Prof. Zuofu Cheng
ZJU-UIUC Institute
May 30, 2023

Contents

1 Introduction to the PokeHead Game

Our main goal for the Final Project is to implement a video game on the Cyclone IV FPGA. We named this game PokeHead, and it is mostly original work from our team, drawing inspiration from various elements and logic found in differengameplayThe primary game play of our game involves controlling a Pikachu to dodge attacks from multiple Gengars, while also accumulating points by defeating them. The game will continue until Pikachu's health is depleted.

The project involves implementation of keyboard, VGA display, RAM and audio. NIOS II CPU is used in order to interact with hardwares.The display of players and enemies and their movements as well as the game logic are implemented in SystemVerilog.

The introduction of the game is section 2 and hardware design details start in section 3.

2 Overall design: Game Logic and Game Elements

2.1 Inspirations and Elements from Other Games

Our game's main logic and mode are directly borrowed from a simple game called BoxHead. In BoxHead, players will control a character to fight against incoming zombies in a 2D role-playing and third-perspective shooter game.

Figure 1: BoxHead Sample

Initially, we tried to transplant BoxHead and its game logic by writing some System Verilog programs. However, due to the low resolution of VGA, we found that the original materials of BoxHead did not display well on the screen. Thus, we came up with the idea of combining elements from other games to design a new game based on BoxHead. Due to the limitation of low resolution, we decided to design a small game with an 8-bit pixel style. This game implementation not only inherits BoxHead's excellent game logic but also displays a unique art style without taking up too much space on the hardware memory.

After careful screening, we have selected the classic early Pokemon as our source of materials, a game originated from the 1990s and was first released on the Game Boy handheld console. There is relatively detailed information and relatively complete animation materials available online. Therefore, we have decided to create a new game based on the materials from Pokemon.

In our game implementation, the attack method and range are slightly different from BoxHead. We have replaced the bullet firing in BoxHead with Pikachu's instant electric shock within a certain straight-line distance to harm enemies. However, the logic and way of the character's movement, avoiding close-range attacks from enemies, and the scoring mode of killing enemies are fully inherited in our game

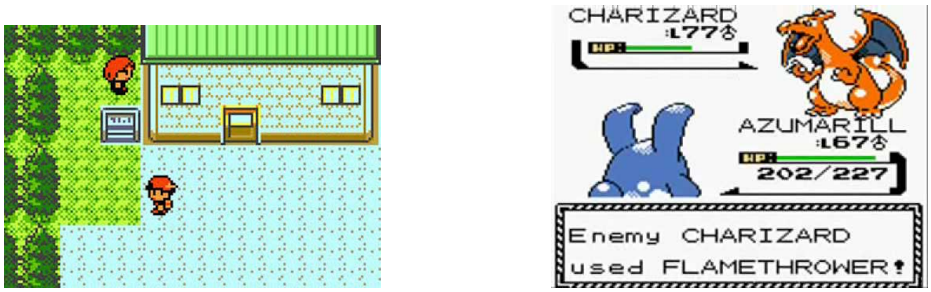


Figure 2: Pokemon Sample

implementation.

2.2 Game Element Player: Pikachu

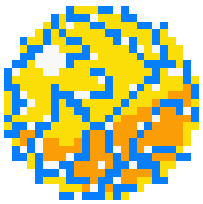
In the following sections, we will introduce the design of the game and different elements of the game. The sprite of player and its attack is shown in figure 3. The basic control logic is given in the table 1. When walking in different directions or attacking, player will show different forms.

We have divided Pikachu's attacks into two forms. The first form is to launch a ball-shaped lightning attack, which is similar to firing a bullet in the third-perspective shooting games. The second form is a area-lightning attack, which deals significant damage to enemies within a certain range in one direction. The second level is unlocked only after you kill a certain number of enemies. These two attacks are controlled by inputting Z and C on the keyboard respectively.

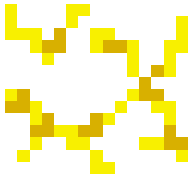
It is worth mentioning that in order to improve the display quality of our game, we adopted a very fancy color palette approach. We will explain and elaborate in detail in the Game Tricks section.

We designed a special electric sound effect for Pikachu's attack, and when Pikachu is attacked, we also imported the attack sound effect of Pikachu making a "Pika" sound.

(a) Pikachu Sprite Demonstration



(b) Ball-Lightening Figure



(c) Area-Lightening Unit

Figure 3: Player Sprite

Function	Keyboard Press
Start Game	Space
Move Rightward	right
Move Leftward	left
Move Upward	up
Move Downward	down
Attack: Ball-Lightening	X
Attack: Area-Lightening	Z

Table 1: Pikachu Control Logic Table

2.3 Game Element Ghost/Enemy: Gengar

Enemy is represent with Gengar in Pokemon. It always tries to follow player and attack. Same as Pikachu, it shows different forms when walking or being attacked. The sprite is shown in figure 5. The features of enemy is listed below.

1. **AI Feature: Greedy Auto-Tracking Agent.** Referring to the game logic of BoxHead, we need to create an enemy agent that automatically tracks the player so as to improve the game logic. We mainly use the greedy algorithm to calculate the Manhattan distance from the player to the four adjacent pixel points in real time, so as to determine the moving direction of the agent. Due to time constraints, we did not design complex maps with obstacles. We only adopted the design of open maps in this game. This map setting also enables our search AI agent to maintain optimal path tracking. The basic process of the Auto-tracking is shown in figure 4.

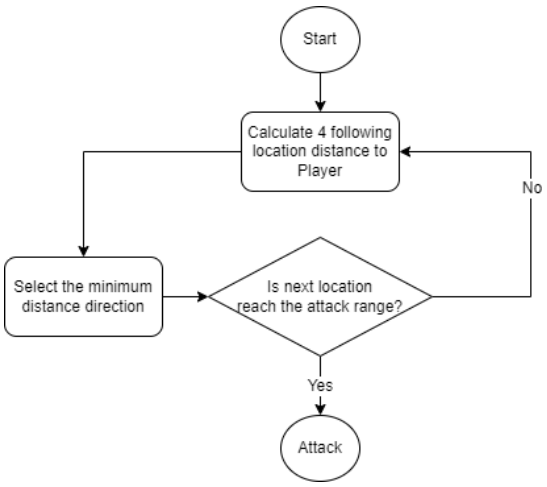


Figure 4: Greedy Auto-Tracking Flowchart

2. **Attack Effects.** Basically, the enemy attack special effects we designed are relatively simple and single. Gengar's attack method is only melee, and only Pikachu played by the player will be attacked within a small range around Gengar. The effect of the attack is to produce purple-black lightning on Pikachu's body. The material used is the same as Pikachu's lightning, but we choose to change the palette to achieve a different display effect.

   Gengar loses his smile, step backwards and stop for a while when he is attacked, we repainted the material from the Internet. The detailed change of the sprite is shown in figure5b and figure5c.

   We also designed the sound effect for Gengar being attacked, and Gengar will make an "uh-ah" sound when he is attacked.

(a) Gengar Sprite Demonstration

(b) Gengar Attacked Sideway

(c) Gengar Attacked Front

Figure 5: Gengar Sprite

2.4 Background and other elements

2.4.1 Background Images

The background construction of the game is based on the simplification of the color tone on the basis of ensuring that the display quality is close enough to the original Pokémon game materials. Since there is no ready-made suitable background, we specially built a Python script to build different backgrounds, and with the help of the course-provided Python script encodes the images to FPGA. The final background is shown in figure 6. It's possible we only need sprite of some elements in the background like floor tile, rocks etc. But for simplicty, we store the whole background image as a whole into ROM.

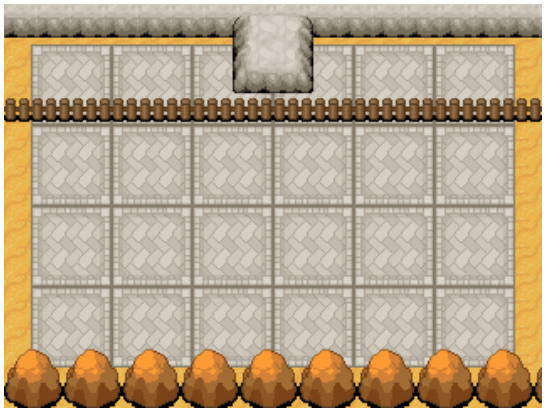


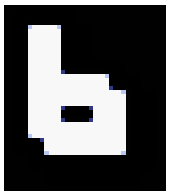
Figure 6: Background Converted After Palette Chosen

2.4.2 Background Music

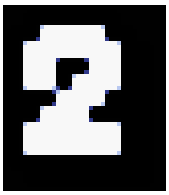
We chose the Littleroot Town theme song as the background music for our game. Littleroot Town (Japanese: ミシロタウン Mishiro Town) is located in southwestern Hoenn, which is shown in figure**??**. It is where players of the Hoenn-based games start their Pokémon journeys. Hoenn's region-exclusive professor, Professor Birch, has a laboratory in the town that he uses for research on Pokémon. Choosing such a piece of music is a tribute to the Pokémon game.

2.4.3 Display of Title, Score, Health etc.

We adopted specially designed sprites rather than using the text mode in lab9. Here are the elements we used to represent various of display needs that the player get. Specifically, we display life of player, score the player gets by killing enemies and current level of the game (The more enemies player kill, the higher level the game is and the more frequent enemies appear). The game start and game over interface is also displayed.

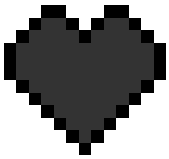


(a) Number Font 6

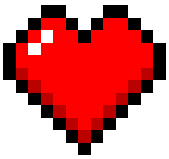


(b) Number Font 2

Figure 7: Number Font Sample

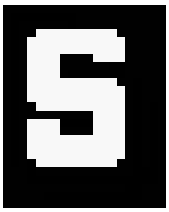


(a) Empty heart

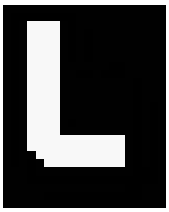


(b) Red Heart

Figure 8: Health Sample



(a) Score Font



(b) Level Font

Figure 9: Font Sample

Figure 10: PokeHead Title

2.5 Game Features and Tricks

In this section, we will introduce all features and tricks of the game.

2.5.1 Sound Effects

We have talked this topic in detailed in the precious section 2.2 and section 2.3. Briefly speaking, we adopted three kind of sound effects: the Background Music, the Attack Music Effects and the Character Music Effects.

2.5.2 AI Feature: Greedy Auto-Tracking Agent

We have introduce the content in section 2.3. Please refer to that section for our AI design features.

2.5.3 Difficulty Level Update

Because our level setting is relatively simple, and there is only one relatively open map, in order to increase the playability of the game, we set the "Difficulty Level Upgrade" function in the game.

Because our game logic is to revive Gengar infinitely before Pikachu's HP is completely exhausted, the way we set the difficulty up is to shorten the cooldown time for Gengar's resurrection. Another operation to improve playability is that players can only use Ball-Lightening to attack Gengar before level 1 and 2, and the more powerful range lightning attack can only be used after level 3 or above. The only way to reach level 3 is to kill as many Gengars as possible.

2.5.4 Cheating Code

As an easter egg of our game, and a tribute to the handheld games and handle games of the last century, we have set "Cheating Code" in the game. The Cheating Code of our game is *up up down down left right left right Z X Z X*.

Because our design is an endless game, if it is set to invincible mode, it will not end the game, so if you presses the key as described, it will increase the blood volume to 30 lives, and increase the speed to twice the original movement speed.

2.5.5 Multiplayer: Competitive Mode

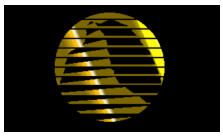
As an easter egg of the game and to explore more possibilities of the game, we added some interfaces to implement a beta version of Competitive Mode.

Our Competitive Mode mainly consists of controlling a Gengar to fight against Pikachu in the normal mode. By pressing G you can toggle this mode. In this mode, another player can control a Gengar using key $W\ S\ A\ D$ and trying to attack Pikachu. But our game logic is still until Pikachu runs out of blood, so in this game, Gengar operated by another player can be resurrected infinitely, so the game is not fair, so we did not explicitly set this mode As a formal mode, it is only used as a test mode in the game and reserves the hardware control interface.

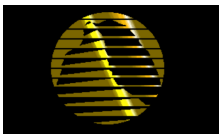
2.5.6 Palette Cycling

Palette cycling is an animation effect that was popular during the days of palettized graphics. It involves rotating the palette colors to give the appearance of movement an otherwise static background. It can be seen most obviously in SCI games in the title screen with the Sierra logo, but also shows up in other places.

Here is a brief example of the effects of Palette Cycling, these two frames are selected from a palette cycling animation.Palette Cycling can make a cool flow effect. We integrated this effect into Pikachu's attack special effects, and it has a very cool lightning flow effect.



(a) Sample Frame 1



(b) Sample Frame 2

Figure 11: Sample of Palette Cycling

3 Written Description of the PokeHead System

3.1 Description of overall design

This project is an extension of Lab8. VGA controller and keyboard part is directly used from lab8. There are four main modules: *player, enemy, attack and enemy attack* and one logic module: *gamelogic*. Player, enemy, and attack's X,Y positions, motion, lives are all stored in registers and they are input to *gamelogic* part. *gamelogic* compares these inputs and determine if a attack causes damages and records lives of player and enemy as well as the score of the game.

There are two clock signal to consider. One is the 50Mhz clock and register is updated after each rising edge of 50MHz. Another is the frame clk, which is set to 15Hz in the game. The change of input values to registers happen only after rising edge of frame clk. When the game is in game start or game over intferface, frame clk is set to 0, causing a frozen screen.

The sprite, background music and sound effect are all stored in On Chip ROM for simplicity. The total usage of OCM is roughly 50 - 60 percent of the total memory

The game also involves usage of technical detailed part in order to make the game more playable. They are listed below.

3.2 Palette

To reduce the size of memory, we use palette to store color. And we notice that the foreground and background have significant different colors so we choose to use two different sets of palette for foreground and background, each with 32 sizes(5 bit). This technique makes background image looks much better. We also use palette cycling for attack so its color keeps changing and makes the game more fancy.

3.3 Finite State Machine

In order to simulate the walking of player and enemies, a FSM is used to indicate different forms of walking. There are four states in total and each state indicate if the player should step its left foot, step its right foot or neither of them. State changes on rising edge of frame clk. The default state is $S0$ and the state stays in $S0$ as long as the motion is 0. Once X motion or Y motion is not 0, state will go to $S1, S2, S3$ and go back to $S0$ again.

Also, the record of cheat code input is also recorded as a FSM. Only after correct key is pressed will the state go to next state. Once all keys are pressed, the cheat code works and God Mode is on.

3.4 Counter

In the game, there is interval between player attack and enemy attack(Cooling Down of attack). There is also different forms of players and enemies when they are attacked. One should notice that these forms last for several seconds. These are implemented using counter that increases at rising edge of game frame clock.

Taking enemy being attacked as a example, the counter is set to 0 be default and it won't change to 1 until the enemy is attacked. Once counter is set to 1, it will keep increasing until it reaches a preset

number. During all this time, the enemy will keeps still. Once the counter reaches the preset number, it will changed to 0 again and wait for next signal to become 1.

3.5 Reuse of Module

The game is endless and there should always be enemies appearing. For each enemy, it is represented as a module. But it is trouble some to create too many modules that are the same even if you can use something like *generate*. So for this game, we only use 4 enemy modules so there are at most 4 enemies being displayed. In order to make the game endless, we keep making enemies respawning. By setting different respawning time for different modules, the game works exactly like an endless playable game.

3.6 Damage Evaluation

The evaluation of enemies are simply determined by the intersection of enemy sprite and attack sprite. The Ball-Lightening attack is easy to determine the intersection but the are-lightening have different X Y Position and width height presentation when player is in different directions. This can be more complicated to evaluate.

For the enemy attack, when a enemy is close to the player, it will stop and change state to *Attack_Ready*. In this state, if cooling down time of the enemy is ready, it will attack the player.

4 Block Diagram

4.1 Design Block Diagram

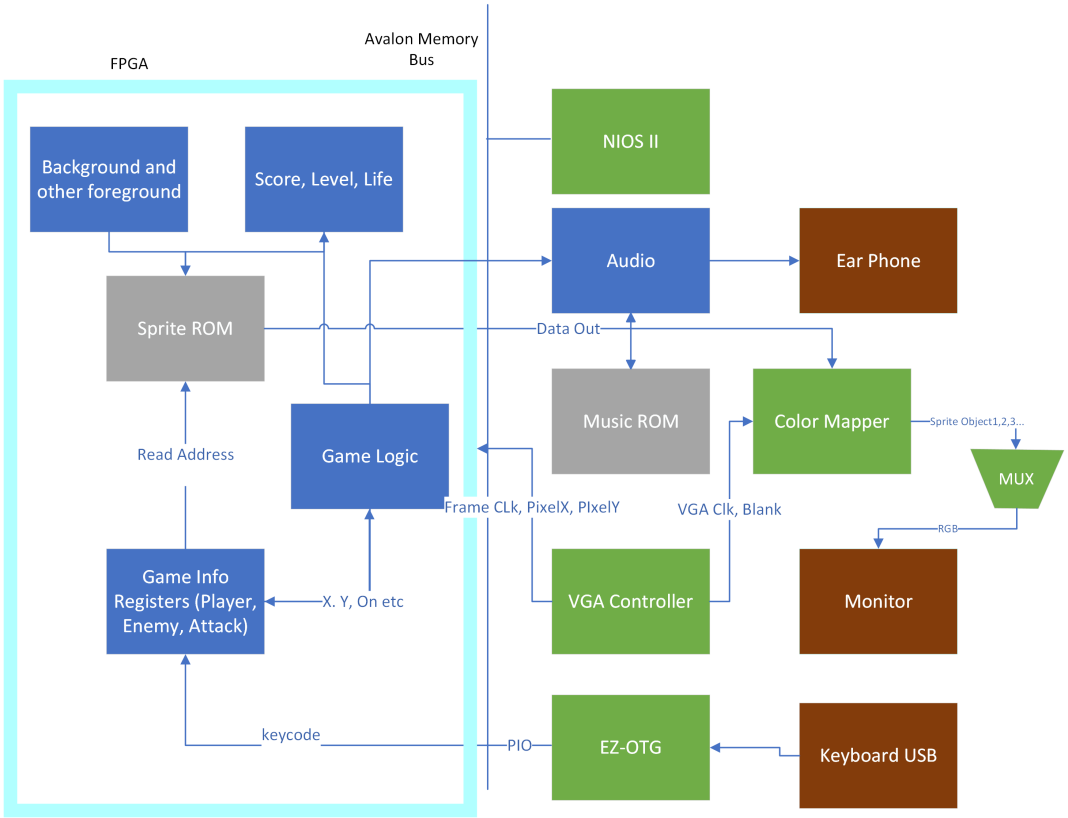


Figure 12: Design Block Diagram

4.2 Platform Designer Diagram

It should look the same as in Lab8

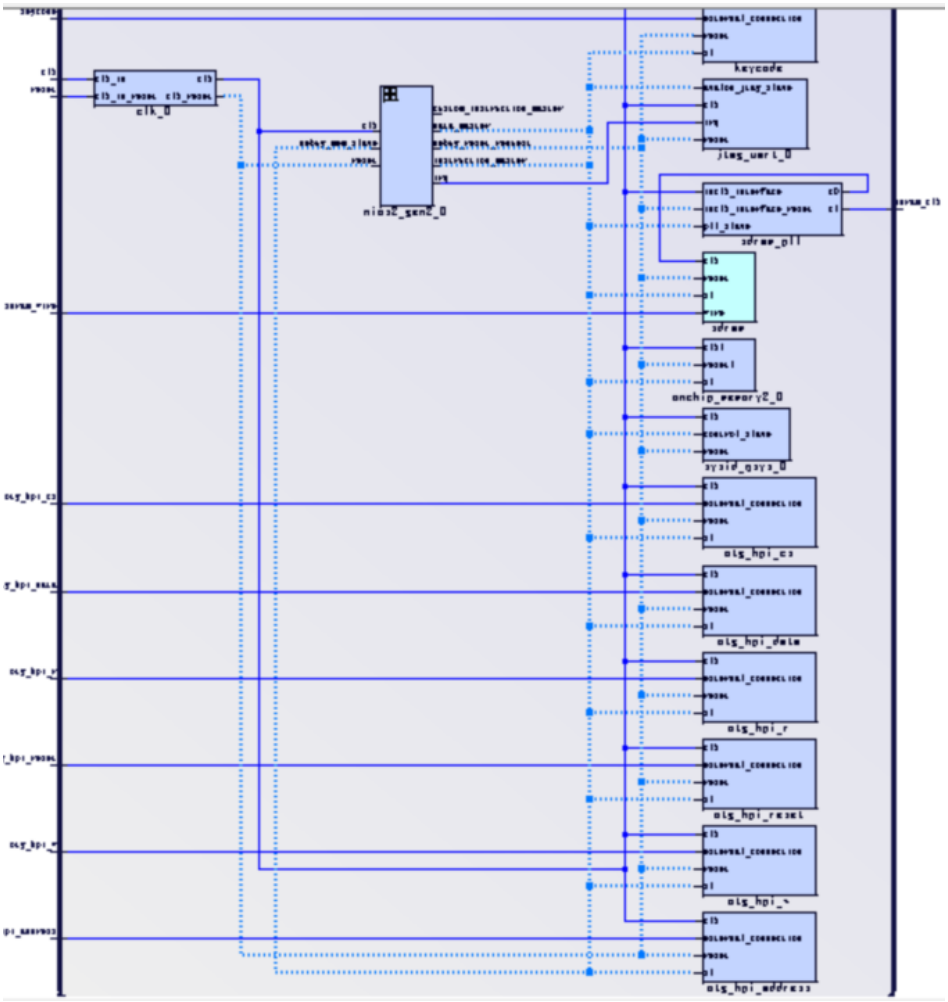


Figure 13: NIOS II Schematic

4.3 RTL Viewer of Toplevel diagram

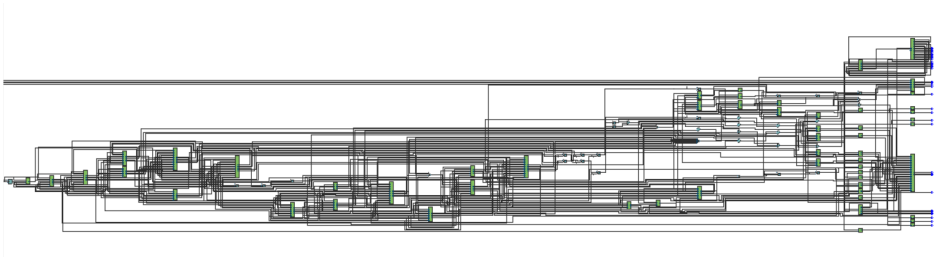


Figure 14: RTL viewer of toplevel

5 Module Description

Some modules like *VGA_controller, hpi_io_intf* are already explained in previous labs and we won't explain here again.

5.1 Game Logic Part

1. **Module**: boxhead
   Inputs: CLOCK_50, KEY, OTG signal, DRAM signal. Audio signal.
   Outputs: VGA signal, HEX, OTG signal, DRAM signal, AUdio signal.
   Description: This module contains all submodules and control the interactions of them. It is directly connected to other devices.
   Purpose: Toplevel of the project

2. **Module**: player, enemy, attack, attack2, blood, gamestart, gamelevel, gameover, score,blackboard
   Inputs: Clk, Reset, game_frame_clk, keycode, PixelX, PixelY, Other_Obj_X, Other_Obj_Y, Other_Obj_On, etc
   Outputs: Obj_X, Obj_Y, Obj_On, is_Obj, Obj_Address, etc
   Description: These are all modules with similar inputs and outputs but different logic. Basically they are all extension of Ball in lab8 but with more complicated logic. Based on current PixelX, PixelY and possbile position of some other objects, it computes whether current pixel drawn is the object and the address of palette index in the OCM.
   Purpose: To control the logic of one element and determine whether the current pixel should be drawn as this object

3. **Module**: playerROM, enemyROM, attackROM, attack2ROM, bloodROM etc
   Inputs: Clk, read_address0, read_address1.
   outputs: data_Out0, data_Out1
   Description: these are all ROMs to store different sprites for different objects. Based on different needs, some ROMs may have one port and some ROMs may have two ports. It simply output data stored in the specified address in the ROM.
   Purpose: To store sprite of different objects.

4. **Module**: gamelogic
   Inputs:Clk, Reset, game_frame_clk, All_Obj_XY, Player_Direction, All_Obj_On etc.
   Outputs: Score, Damage, Is_Attacked etc.
   Description: takes all necessary inputs and evaluate the damage of a attack, records player's lives and player's score. It also records if a enemy is alive and whether it has respawned.
   Purpose: serve as the game logic element.

5. **Module**: player_control, enemy_control
   Inputs: Clk, Reset, game_frame_clk, Obj_XY_Motion
   Outputs: Obj_Step_Count.
   Description: finite state machines to record current step of the object so that the objects shows different status when walking.
   Purpose: To make sure objects show different steps when walking.

6. **Module**: godmode
   Inputs: Clk, Reset, game_frame_clk, keycode
   Outputs: God_Mode_On.
   Description: A FSM that changes state based on keycode. when certain keys are pressed, it goes to final state and god mode is on (cheat code works).
   Purpose: To judge if god mode is on.

7. **Module**: color_mapper
   Inputs: Clk, is_obj, obj_index
   Outputs: VGA sigal
   Description: Based on if current pixel is object and its palette index, assigns color to it. It judges between differetn foreground and also achieves palette cycling.
   Purpose: To assignal color output for VGA.

8. **Module**: game_frame_clk
   Inputs: Clk, frame_Clk, Game_Start_On, Game_Over_On
   Output: game_frame_clk_rising_edge
   Description: based on original VGA clock and if game start or over, assign frame clock for the game.
   Purpose: provide game frame clock.

5.2 Audio Part

1. **Module**: audio
   Inputs: Clk, Reset, INIT_FINISH, data_over, music_frequency
   Outputs: INIT_BGM, Add
   Description: A Finite State Machine, this module is to generate the pointer address to the on chip memory. which also means that it can modify the music play speed and control whether to play the music or not.
   Purpose: Give music module the address, control the play speed and control when to stop playing music. Playing background music.

2. **Module**: music
   Inputs: Clk, Add
   Outputs: music_content
   Description: Based on ROM content reading approach, we implemented a series of On-Chip Memory to store the music content.
   Purpose: Given the address generated by the Audio module, we return back the music content value of certain point to the port music_content.

3. **Module**: "XXX"_audio
   Inputs: Clk, Reset, INIT_FINISH, data_over, "XXX"_enable, "XXX"_frequency
   Outputs: INIT_"XXX", "XXX"_add
   Description: A Finite State Machine, this module is to generate the pointer address to the on chip memory. which also means that it can modify the music play speed and control whether to play the music or not.
   Purpose: Give music module the address, control the play speed and control when to stop playing music.

4. **Module**: "XXX"_sound
   Inputs: Clk, "XXX"_add, "XXX"_enable,
   Outputs: "XXX"_content
   Description: Based on ROM content reading approach, we implemented a series of On-Chip Memory to store the music content. And there is an enable port which ensure that if there is no audio playing, there will be no potential output sound.
   Purpose: Given the address generated by the Audio module, we return back the music content value of certain point to the port music_content.

6 Demonstration of PokeHead Game

(a) Game Screenshot 0

(b) Game Screenshot 1

(a) Game Screenshot 2

(b) Game Screenshot 3

7 Conclusion

7.1 Design Statistics Table

LUT	40791
DSP	8
Memory(BRAM)	2,368,512
Flip-Flop	2924
Frequency(Mhz)	134.01
Static Power(mW)	104.62
Dynamic Power(mW)	0.94
Total Power(mW)	193.96

Table 2: Design Statistics Table

7.2 Functionality of the PokeHead Game

The game works with resolution 320×240 pixels and at frame rate 15Hz. The player is controlled by the keyboard and enemies always tries to follow the player. Player and enemies show different forms when walking and the color of attack changes as well. The game supports VGA display, multi keyboard input, audio output all well. The game difficulty increases as player kills more. The game also has a good game start and game over interface. Sometimes, the damage of attack may not work very well, especially when the game level is high (possible due to short time of respawn). The audio can be improved more. Overall, the functionality of the game is good and is quite playable.

7.3 Challenges Encountered in Project

1. Game Asset Organization Issues

   In the process of designing the game, we found many very interesting and high-quality sprites and background materials. But the challenge we encountered was that the materials we found often did not match, that is to say, there were always some unsatisfactory places in the display of characters and backgrounds. For example, the perspective angle and size relationship of the materials we found are different from the ideal background-character combination. To overcome these differences, many background elements and some sprites are completely hand-painted and manually modified by ourselves.

2. The balance between On-Chip Memory space and image and audio quality

   Since the game logic and real-time sound effects of our project are all built using hardware circuits, we need very high-speed sprite and audio reading and writing performance. In terms of graphics display, we must limit the number of colors within a range to ensure that the graphics will not occupy Large capacity of On-Chip Memory. In terms of graphics processing, we used three sets of different Color Palettes, one for background generation, one for sprite generation, and the last set for color cycling of attack effects. Such processing greatly reduces the space occupied by the image. Similarly, we downsample the audio, reducing the sampling frequency while ensuring the pitch of the audio is correct.

3. Distinguish between Clk and $frame\_clk$

   Sometimes it is easy to confuse Clk with $frame$ and you don't know what clk signal you should use. For example, when you read keycode, should you update status only after next frame or you should update it now? For the counter, how can you make sure it reaches 0 when it satisfies certain condition.

4. Complicated forms of objects in the game.

In the game, both player and enemies have four directions and different forms in four directions. The direction of the player will affect the direction of attack as well. So you need to evaluate different conditions for different directions. Sometimes this can be error prone and not easy to debug.

7.4 Further Improvements

The game has great extendability and can involve lots of features. Here we highlight a few points.

1. Support of more enemy display

   For now, the game only supports display of four enemies but this can be extended to display with simple modification. More enemies can increase the playability of the game.

2. Use SDRAM or SRAM instead of OCM only

   This project use OCM only for simplicity. Too much use of OCM will make the compilation time too long. It can be better if we use some part of SDRAM or SRAM to store memory instead.

3. Add obstacle and screen scrolling

   These are all possible things that we can add to make the game more interesting.

4. Support more devices

   For example, we can try to support disk storage to keep a record of the highest score. Or we can support Ethernet to support multiple player online. Or support mouse to do something interesting.

5. Add wiser AI auto-tracking agents

   The current AI auto-tracking agent is just using greedy algorithm approach, but if there are more complicated maps, the agent may not be optimal, if we could implement a wiser agent, the game would be much more interesting.