Hardware Design Final Project

FPCAT – Battle Cat on FPGA

Team 01

Sheng Wei Fan 112062144
Chia Chin Hsieh 112062122

Table of Contents:

1. Introduction

1.1 Motivation

The Battle Cats has been my favorite mobile game since I was in elementary school.

▲ Figure 1.1: An Instagram post of Chia Chin from 5 years ago

The stage and character designs are absolutely top-tier, even by today's standards. To share the fun of playing The Battle Cats with more people, we plan to create our own version of the game using FPGA. By showcasing this project on GitHub, we hope to demonstrate the game's appeal.

1.2 Overview

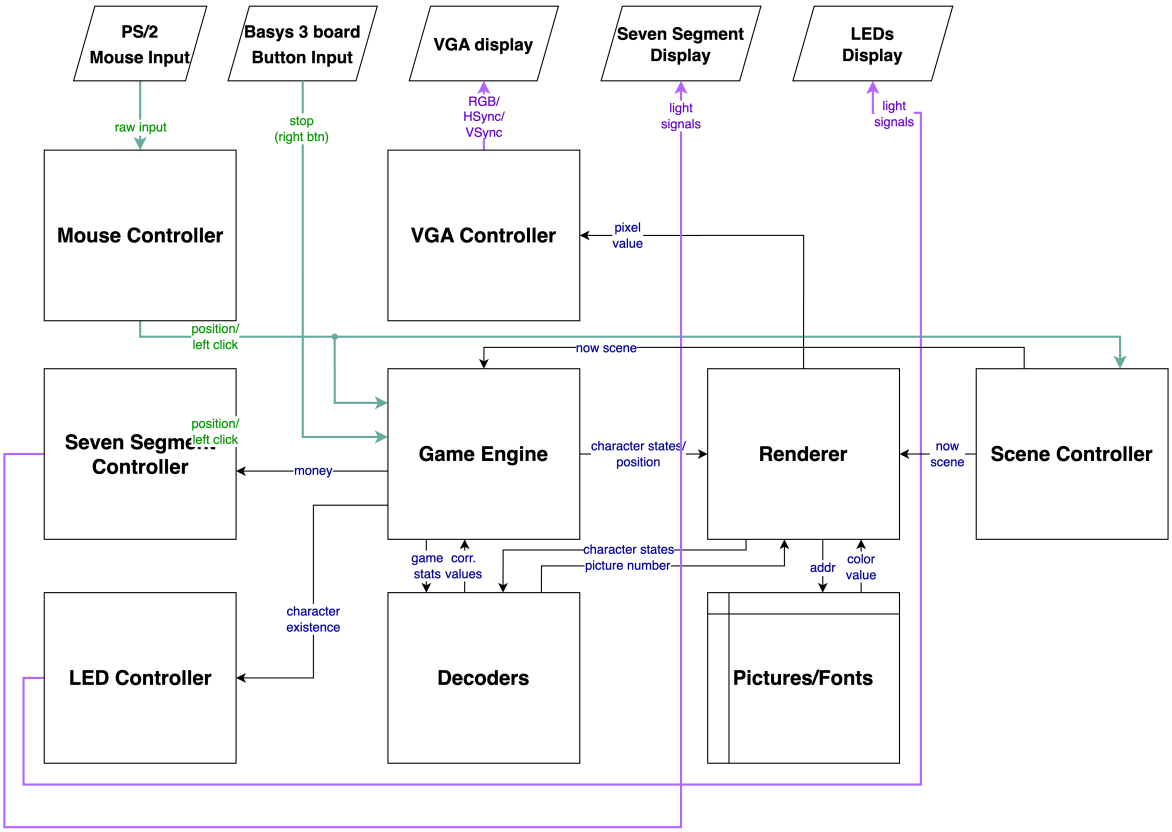
Our inspiration for the name "FPCAT" came from the integration of "FPGA" and "CAT". Our game requires players to use the mouse to operate. For screen signal transmission, we use VGA standard. The game begins with a **start scene**. After pressing the **start button**, the game transitions to the **menu scene**. each with designated enemies assigned to it.

In the **game scene**, players will have access to buttons for **cats**, **tower**, **purse**, and **pause** (operated via an FPGA button). To **summon cats**, players must accumulate enough money, which can be earned over time. The **tower** can emit air cannon after it finishes charging. Players can also spend money to expand the **purse capacity** and speed up the accumulation of money.

The goal is to destroy the enemy tower using the player's cats. If the player's cats succeed, the player wins. If the enemy destroys the player's tower, the player loses.

After triggering the **win scene** or **lose scene**, players can return to the **menu scene** by clicking the mouse.

The figure below illustrates the module structure and relationships within our project. The signal painted in green represents the input of our Top module, while the signal painted in purple represents the output, and the remaining blue signals are important signals transmitted in out project.



▲ Figure 1.2: Top Abstract Block Diagram

This is a brief overview, with details discussed in the following chapters. For example, the renderer consists of four smaller modules: Render_Start, Render_Menu, Render_Play, and Render_WinLose. Each render module computes its own color code independently. This is why it needs to take input from the scene controller's output. The details will be covered in Chapter 5.

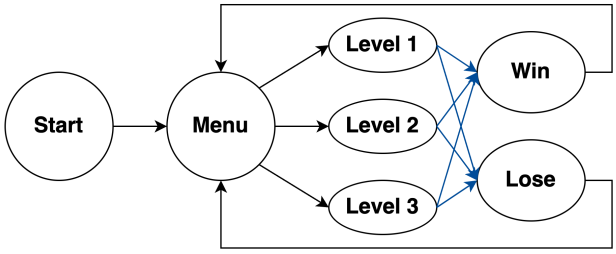
As for the Game Engine, we have represented it as a single block in the diagram. However, it is one of the most complex components we have implemented. It requires four different clocks and multiple signals to function properly. Additionally, it manages various gameplay elements, such as character states and game stats updates. Therefore, designing robust FSMs (Finite State Machines) is critical. We will elaborate on this in Chapter 4.

Finally, we have added some thoughtful touches to the picture and font components, which will be discussed in Chapter 3.

2. Game Scene Architecture and Implementation Details

This chapter introduces our game's scene design and its detailed components. The following sections will explain our implementation methods, challenges we encountered during development, and how we solved these problems.

First, here is our scene's FSM (Finite State Machine):



▲ Figure 2.1: Scene's FSM

The scene transitions marked in blue are the only ones triggered by the game engine itself, while all other transitions are initiated through button interactions. All mouse click detection implementations include **Debounce** and **One Pulse** mechanisms to ensure reliable input processing. For consistency, we used the `clk_25MHz` for these detections, which is the same clock signal used for VGA output and scene transitions.

2.1 Start and Menu Scene

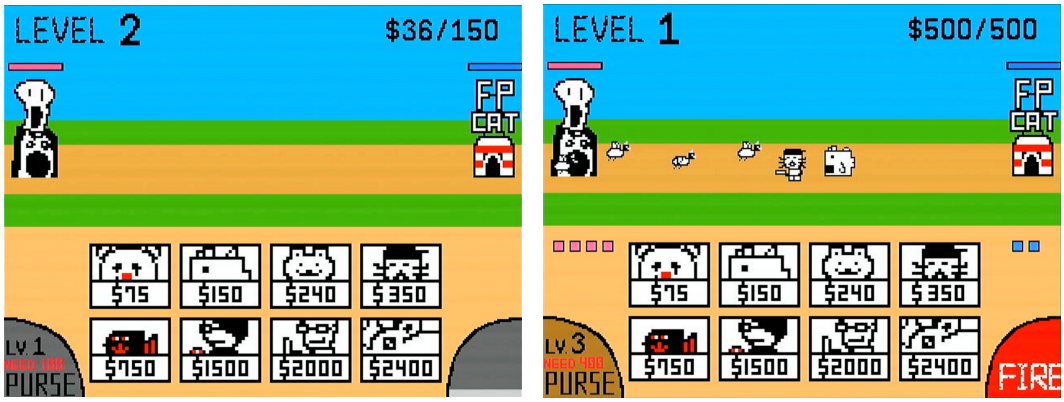
We designed separate Start Scene and Menu Scene components based on our reference game Battle Cat, which features an initial game start button before entering the main menu. We decided to keep this design element as it provides a clear game entry sequence for players.

▲ Figure 2.2: Screenshot of Start and Menu Scene of ours and Battle Cat

Additionally, both scenes feature interactive buttons with hover effects - when the mouse cursor hovers over a button, its color changes to provide visual feedback to the player.

2.2 Play Scene

Below is our game interface. We will explain each implemented feature in detail.



▲ Figure 2.3: Screenshot of Play Scene

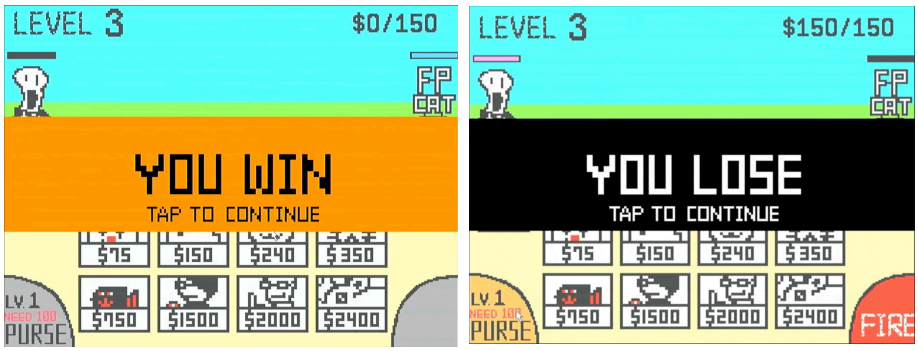
- The Stage number is displayed in the top-left corner, indicating the current level.
- The top-right corner shows both our current money and maximum money capacity.
- In the bottom-left corner is the purse upgrade button. The Purse Level affects both the money generation rate and the maximum money capacity. The button appears gray when there's insufficient money for an upgrade and brown when an upgrade is available. The button displays both the current level and the cost for the next upgrade. The current money will also be displayed on Seven Segment Display.
- The bottom-right features the tower firing button. When fully charged, it shows a red "ready-to-fire" state. After firing, it turns dark gray and gradually transitions to light gray in segments as it recharges. This charging mechanism pays homage to the original Battle Cat game.
- If pause switch is turned on, it'll show a red block at the top to indicate the halt.
- Our tower is positioned on the right side with its HP displayed in blue, while the enemy tower is on the left with HP shown in red.
- The character deployment buttons are located at the bottom of the screen. Unit deployment is only triggered if sufficient money is available. When a unit is deployed,

money is deducted and a cooldown effect appears on the button to prevent rapid consecutive deployments of the same character.

- Enemy unit deployment follows a pre-designed Enemy Queuing system, which will be detailed in Chapter 3.2.
- The red and blue little blocks on the sides of the screen indicate the current number of enemy and army (our cats) on the battlefield respectively (this information is simultaneously displayed on the FPGA's LED display). Both sides are limited to a maximum of 8 units on the field at any time. When this limit is reached, no additional units can be deployed. These frames serve as visual indicators, flashing when the 8-unit limit is reached to alert players that they cannot deploy more units, or that enemy unit deployment has been temporarily paused. When one or more enemy units are defeated, multiple new enemy units may be spawned simultaneously, potentially creating challenging combat situations.
- The battle between both sides takes place on a one-dimensional line. However, to prevent complete overlap of all unit layers during multiple-unit combat, units are randomly offset upon generation. All attack and detection calculations remain one-dimensional. This mechanism aligns with the original game Battle Cat.

2.3 Win and Lose Scene

For these two scenes, we overlay a horizontal bar on the game screen displaying the win/loss text. A flashing "Tap to Continue" prompt is added to inform players they can return to the Menu Scene. Notably, in addition to the horizontal bar, we invert the colors of the final game screen behind the bar to emphasize it.



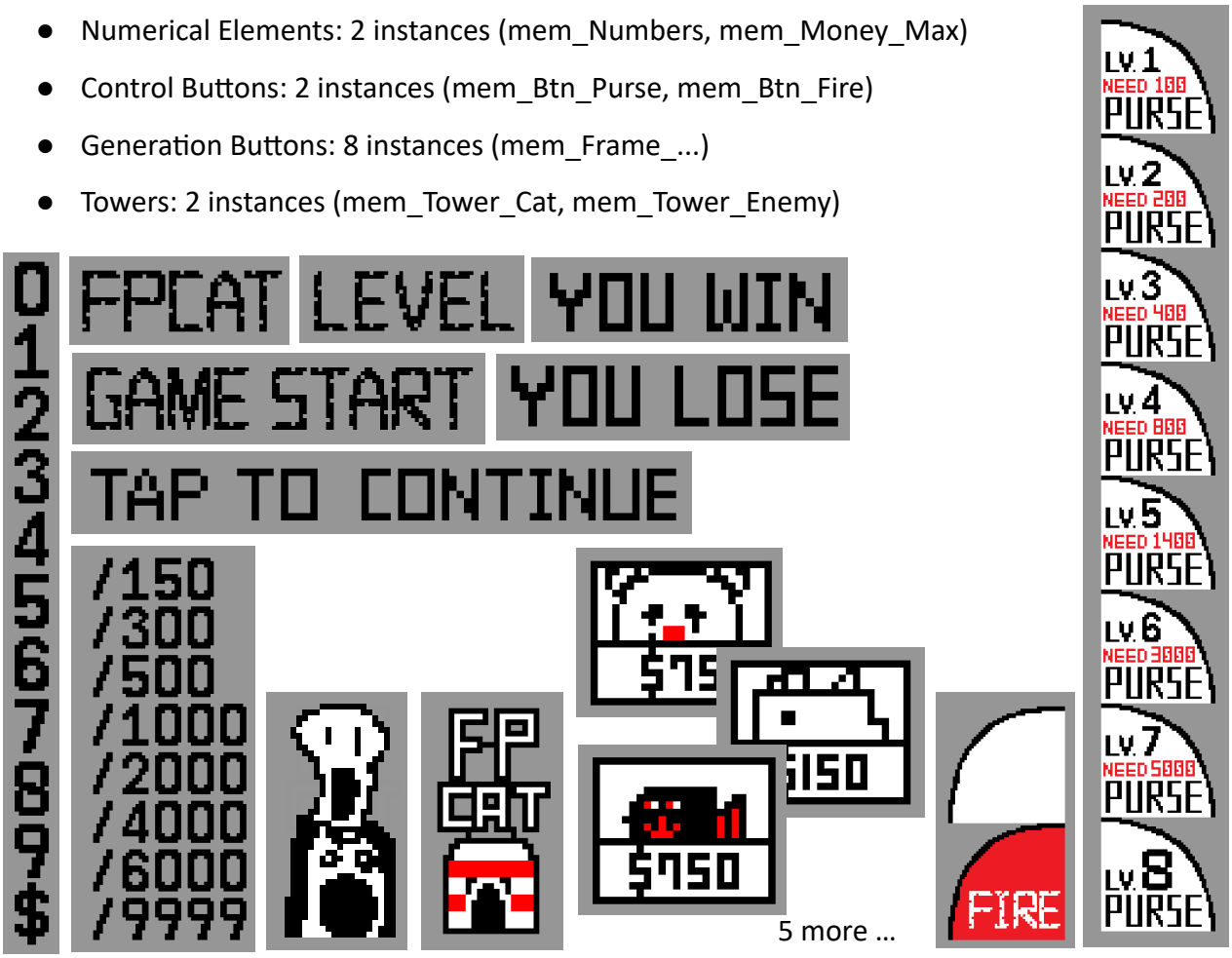
▲ Figure 2.4: Screenshot of Win/Lose Scene

3. Game Material Design

3.1 Static Object Design

We utilize IPs (Single-port RAM) to store 19 Static Objects, including:

- Text Elements: 6 instances (mem_FPCAT, mem_LEVEL, mem_GAME_START, mem_YOU_WIN, mem_YOU_LOSE, mem_TAP_TO_CONTINUE)
- Numerical Elements: 2 instances (mem_Numbers, mem_Money_Max)
- Control Buttons: 2 instances (mem_Btn_Purse, mem_Btn_Fire)
- Generation Buttons: 8 instances (mem_Frame_...)
- Towers: 2 instances (mem_Tower_Cat, mem_Tower_Enemy)



▲ Figure 3.1: Self-Designed Static Objects

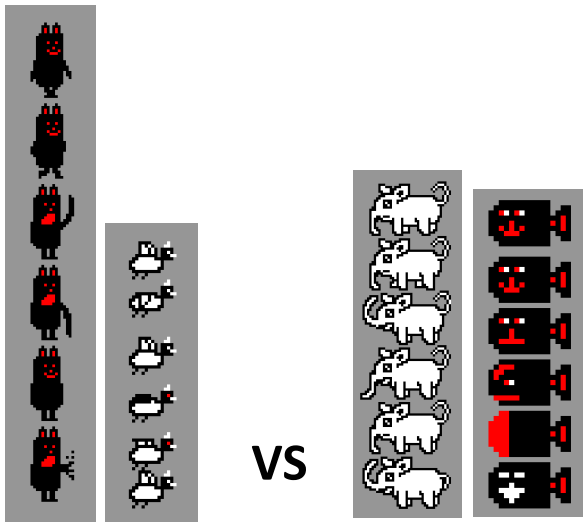
These elements were all designed by us and converted to **coe** files using Python scripts (See Appendix A.1 for details) to serve as initial values for the RAM modules (these Static Objects' RAM values remain constant and are used solely for rendering purposes).

To optimize storage requirements, all coe files storing images, including the characters mentioned in Chapter 3.2, were converted from PNG format using two methods. The first

method, applied to single-color images, uses one bit per pixel to represent either colored or transparent states. The second method, used for three-color images such as characters, towers, and buttons, employs two bits per pixel to represent four states: black, white, red, and transparent. These approaches reduce the storage requirement from 12 bits per pixel (RGB) to either 1 or 2 bits per pixel.

3.2 Character Design

We designed 6 images for each character, corresponding to five states: the first two represent the move state, the third to fifth correspond to atk0-1, atk2, and atk3, respectively, and the sixth represents the repel state. A decoder determines the image to access based on the character's state. Since character heights vary, another decoder called Pixels can be used to retrieve width, height, and difference (used for range detection) based on character type. The picture size (height × width) is multiplied by the picture number to calculate the correct address, which is then used to render the character's state via the IP memory module.

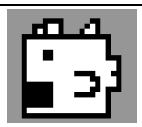
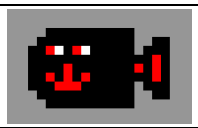


▲ Figure 3.2: Example for 6 Images for Each Character

While all characters in the game are original designs, some pay tribute to Battle Cats. For example, the enemy Black Bear is inspired by a classic character from the original game, known for its speed, high attack, and low health. In FPCATS, it's less powerful to reduce game difficulty. Similarly, Bomb Cat is a nod to "Express Cat," sharing its high speed and attack but with slower attack speed.

We also added CS-themed characters like Elephant Cat (a classmate's nickname), Hacker Cat, and CY Cat (named after a professor).

Below are all the character illustrations we designed:

Army (Cats)				
	Joker Cat	Fish Cat	Trap Cat	Jay Cat
				
	Bomb Cat	CY Cat	Hacker Cat	Elephant Cat
Enemy				
	Killer Bird	White Bear	Metal Duck	Elephant Cat

▲ Table 3.1: Character Illustrations

Here is some short introduction of each of the characters:

- Joker Cat: a mass-produced disposable unit with low health but very low cost.
- Fish Cat and Trap Cat: mass-produced tank units with high health and relatively low cost.
- Jay Cat: a long-range damage dealer with low health but decent range and attack power.
- Bomb Cat: an Express Cat having high speed and high damage but with slower attack speed and lower health
- CY Cat and Hacker Cat: versatile mid-range damage dealers. While their range isn't the longest, their attack power is very high.
- Elephant Cat: a tank damage dealer. It has a longer range than mass-produced tanks, very high health, and decent attack power.
- Killer Bird: a standard small enemy, similar to goblins in an isekai setting.
- White Bear: a typical monster that often gives beginners a tough time.
- Metal Duck: a boss-level enemy.
- Elephant Cat: a tanky damage dealer. It has a longer range than mass-produced tanks, very high health, and decent attack power.

Below are the stats for FPCATS characters:

	Joker Cat	Fish Cat	Trap Cat	Jay Cat	Bomb Cat	CY Cat	Hacker Cat	Elephant Cat
HP	300	600	1500	400	300	800	1400	3500
ATK	30	50	40	140	280	200	180	100
ATK CD	8	8	15	12	15	12	12	15
Speed	2	1	3	3	8	1	1	1
Range	15	10	5	40	5	35	35	20
Price	$75	$150	$240	$350	$750	$1500	$2000	$2400

	Killer Bird	White Bear	Metal Duck	Black Bear
HP	400	1500	4000	140
ATK	40	140	100	80
ATK CD	10	15	15	3
Speed	1	1	1	8
Range	10	20	10	20

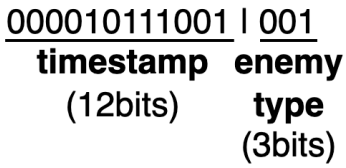
▲ Table 3.2: Character Statistics

Speed is measured in pixels/`clk_6`, range in pixels, and atk_cd in `clk_6` units. For detailed specifications, please refer to Chapter 4.

It is noteworthy that while the Black Bear's attack power appears low, its short atk_cd results in a DPS (damage per second) that is four times higher than that of the Metal Duck.

3.3 Stage Design

To summon enemies using the same mechanism as Battle Cats, we use **Enemy Queues** storing in IPs (Single-port RAM), each record's value of which contains a **timestamp** (12 bits) and an **enemy type** (3 bits). Details will be discussed in Chapter 4.



▲ Figure 3.3: Enemy Queue Protocol

Our first stage (Level 1) is designed as a simple demonstration. Each type of enemy is summoned sequentially with large time gaps in between.

The second stage (Level 2) is a more formal challenge. Shortly after the level begins, Metal Duck will appear. Players will need to use meat shields or fodder units to buy time and save money. In the mid-game, several White Bears will spawn. If the player doesn't deal with the Metal Duck in time, the difficulty in the later stages will significantly increase. In the late game, Black Bears will start appearing to clear the field. Players must quickly eliminate them to avoid being pushed back too far.

The third stage (Level 3) is the most challenging level. The enemy spawning mechanism is similar to that of Level 2, but the time intervals are shorter, and there are more Metal Ducks and White Bears. Just like in Level 2, Black Bears will appear in the late game to clear the field. If they are not dealt with in time, the player will likely lose very quickly. Unlike Level 2, in Level 3, if players fail to manage their economy properly at the start, they don't need to worry about the late game being too difficult — because they'll probably lose in the early game.

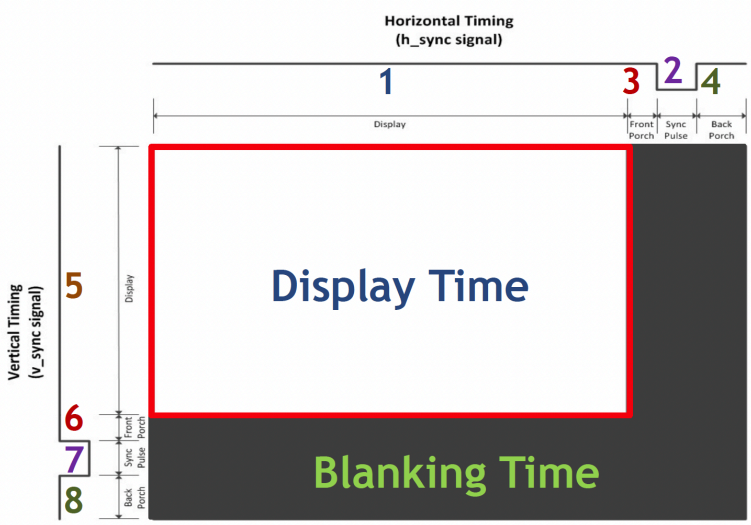
4. Game Engine

First, we will explain the main clock signals utilized in our system:

Signals	Relationship	Descriptions
`clk`	from FPGA	System Clock (100MHz) - This is primarily used to generate the 25MHz clock signal.
`clk_25MHz`	clk / 4	Our main operational signal is the clock required for VGA to generate 640x480 60Hz output with buffer time. During the rendering phase, each pixel's color code is designated within one this clock cycle. Consequently, this clock signal is used for both our game engine operations and mouse input detection.
`clk_frame`	clk_25MHz / 800	This is a signal indicating a frame update, which is mainly used to generate `clk_6`.
`clk_6`	clk_frame / 9	The signal was initially named `clk_6` as we aimed to achieve 10 updates per second, which would be 60Hz divided by six. However, during game testing, we found this frequency to be too rapid, so we modified it to divide by nine instead. This frequency-divided signal is primarily used for Game Engine updates, facilitating the management of character movement and attack intervals. It serves as the main update clock for counters within the Game Engine.

▲ Table 4.1: Clock Signals

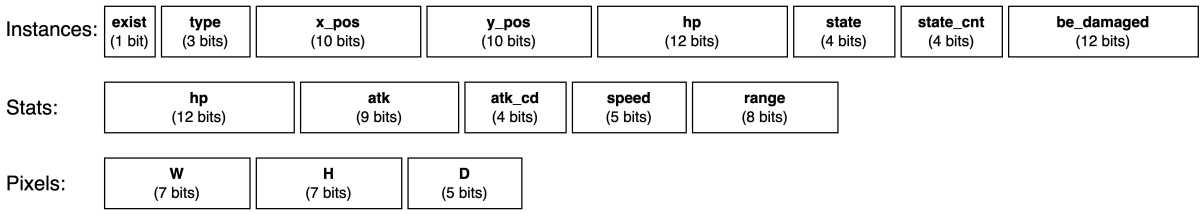
In the game scene, we divide the game process into two parts based on VGA protocol characteristics. VGA requires blanking time at the end of each line to pull back the pointer to the line's beginning, and corresponding blanking time after completing an entire frame. In our project, while the final output is 640x480, the `pixel_cnt` and `line_cnt` in the VGA Module run up to 800x525. We utilize this non-output time period, specifically when `line_cnt` equals 490, to start processing the game engine operations.



▲ Figure 4.1: VGA Sync Process

4.1 Storage Protocols

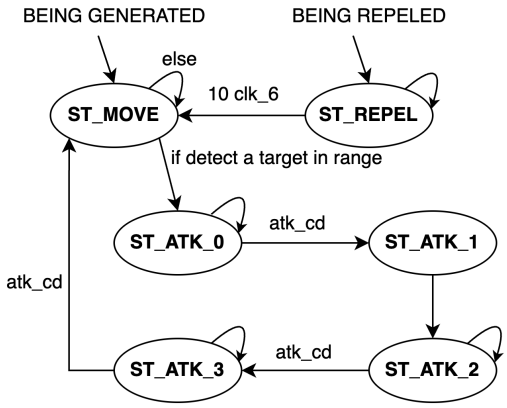
First, we'll introduce the main protocols used. In addition to the previously mentioned three Enemy Queues that use 15 bits for timestamp and 3 bits for enemy type, below are several key protocols. Instances are implemented as a register, with Army and Enemy each having space for 8 Instances. The other two components, Stats and Pixels, are decoders that output corresponding data based on the input Army/Enemy Type (3 bits).



The `state`, `state_cnt` and `be_damaged` in **Instances** will be explained later in this chapter. **Stats** information has been previously explained in Chapter 3.2. **Pixels** stores each character's Width, Height, and Difference values. Difference represents the offset from the image boundary to the character's actual body, used for precise range detection, which will be explained in Chapter 4.3.

4.2 Character FSM

Before explaining the overall Game State operation, we'll introduce the character FSM.



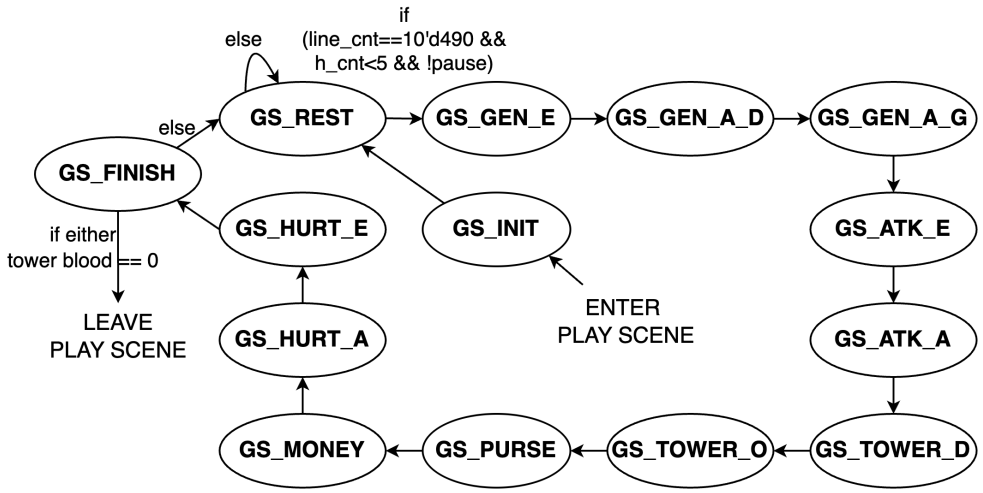
▲ Figure 4.2: Character FSM diagram

Each character has 6 states, beginning in `ST_MOVE` state upon generation. In this state, the character moves a specified number of pixels based on its speed value during each game engine update (`clk_6`). When detecting an enemy within its attack radius, it enters `ST_ATK_0` and waits for `atk_cd` clock cycles before transitioning to `ST_ATK_1`. Both these states use the third image, representing the attack wind-up animation. `ST_ATK_1` lasts for only one clock cycle and executes attacks on opponents, including towers, within the attack radius. The attack is recorded in the target's `be_damaged` field. Subsequently, the character transitions through `ST_ATK2` and `ST_ATK_3`, using the fourth and fifth images respectively, representing the attack animation and follow-through. Finally, it returns to `ST_MOVE`, awaiting movement or the next attack sequence.

Operating independently from this cycle is `ST_REPEL`, triggered by two events: when the character's health drops below half, or when an Enemy is hit by our air cannon. During these ten clock cycles, the character is knocked backwards, after which it returns to `ST_MOVE`. Besides, the mechanism of knockback upon receiving damage at half health is also derived from the original game Battle Cats. This feature not only allows for strategic manipulation of battle lines through enemy knockback but also serves as an indicator of a character's approximate current health status.

4.3 Game State FSM

This chapter explains the core operation of the game system. As previously mentioned, the game engine detection timing is based on the `clk_25MHz` signal with a `clk_6` frequency divider, which determines the update frequency for character movement, attacks, money accumulation, tower cooldowns, and other game mechanics. The game's core functionality relies on the interactions between `Army_Instance`, `Enemy_Instance`, Enemy Queue, and various buttons. Below is the FSM Diagram of our Game State.



▲ Figure 4.3: Game State FSM diagram

Next, we will explain the operation of each Game State:

- **GS_INIT**: When a player selects a Stage to enter the game, it first enters this state. Within one clock cycle, it initializes all signals, including resetting the exist flag of both sides' Instances to 0, restoring both towers to full health, resetting money to zero, etc., before proceeding to `GS_REST`.
- GS_**REST**: This is the state where the system remains after completing a game operation round. It performs no actions and waits for the next round to begin, which occurs only when `clk_6` is active, `line_cnt` reaches Blanking Time, and the pause switch isn't activated. This implements the pause functionality, which is particularly crucial for experienced Battle Cats player Chia Chin, as it allows players to pause and strategize.

- **GS_GEN_E** (Generate Enemy): This state generates enemies according to the designed Enemy Queue. The game uses a `game_cnt` as a counter for the overall game progress and a pointer indicating the next generation line. When `game_cnt` matches the timestamp pointed to by the pointer, it searches for available positions in `Enemy_Instance`. If a position is found, it generates a new entity based on the `Enemy_Stats` Decoder output and moves the pointer to the next line. If all eight entities exist, no generation occurs and the pointer remains static until the next detection when a position becomes available.
- **GS_GEN_A_D** (Generate Army Detection): This single clock cycle state determines whether to generate ally characters (cats), checking for button clicks, sufficient money, and completed cooldown times. The result is passed to the next game state.
- **GS_GEN_A_G** (Generate Army Generation): Receives generation requirements from the previous state. If none exist, it advances to the next state in the following clock cycle. If requirements exist, it searches for available generation positions. Upon finding a position, it deducts money and initiates the button cooldown. If no position is found, the generation attempt is nullified and no action will be applied.
- **GS_ATK_E**, **GS_ATK_A**: These states update each character's state based on their current character state, such as `ST_MOVE`'s detection for attack or movement. Range detection utilizes `Pixels`' Difference for consistent attack range starting points, ensuring proper front-back relationships during combat as designed. Other aspects have been described in Chapter 4.2 and won't be repeated here.
- **GS_TOWER_D** (Tower Detection): Similar to `GS_GEN_A_D`, it checks if the tower's fire button is pressed and cooldown has ended, passing results to the next state.
- **GS_TOWER_O** (Tower Operation): Sequentially checks `Enemy_Instance` for entities within the tower's fire range. If found, applies damage and changes their character state to `ST_REPEL` for knockback effect.
- **GS_PURSE**: Checks if the Purse Level upgrade button is pressed, funds are sufficient, and current level hasn't reached maximum. If all conditions are met, deducts upgrade cost and increases Purse Level.

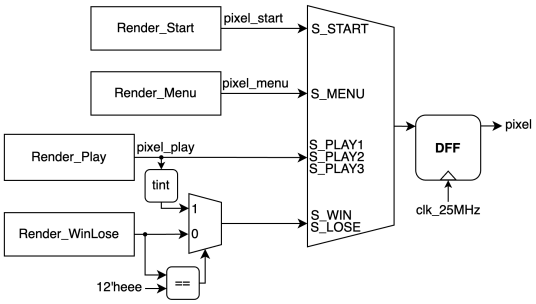
- **GS_MONEY**: Simply increases money according to the current Purse Level.
- **GS_HURT_A**, **GS_HURT_E**: Applies the `be_damaged` values recorded in each Instance to their `HP` during this update round. If the `be_damaged` exceeds remaining `HP`, sets `exist` to zero, resulting in entity death.
- **GS_FINISH**: Finally, when a round completes and returns to this state, it checks tower health. If either tower's health reaches zero, indicating a decisive outcome, exits the play scene and enters win/lose scene. Otherwise, enters `GS_REST` awaiting the next update round.

At this point, we have explained our main game process implementation. Next, we will describe how we render these elements on the display.

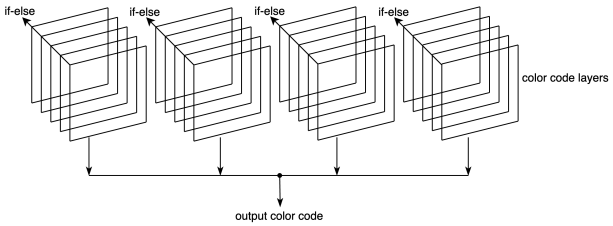
5. Graphics Rendering

5.1 Rendering Module Architecture

As mentioned in Chapter 1.2, our rendering module is divided into four components: Render_Start, Render_Menu, Render_Play, and Render_WinLose. Each component calculates its pixels independently, and the final pixel value is multiplexed based on the current scene. Below are the Renderer's Abstract Block Diagram and visual representation.



▲ Figure 5.1: Abstract Block Diagram of Renderer



▲ Figure 5.2: Visual Representation of Renderer

Notable is our implementation of the Win/Lose Scene display. Since we want to preserve the game's background at the moment of completion, Render_WinLose only outputs the required color code for the central banner, filling other non-rendered areas with an arbitrary color code (`#eee`). Before final VGA output, areas with `#eee` are replaced with the original Play Scene rendering, with an added white tint background effect. This emphasizes the central banner and signifies game completion. This effect is achieved by increasing all RGB values of the Render_Play color code output to create a brightening effect.

The implementation of our Renderer differs from the VGA template used in Lab 6 this semester. In the original template, `pixel` value was directly fed to the VGA output. However, since our Render Modules require certain computational overhead, using a purely combinational approach as output would result in unstable signals at the beginning of each `clk_25MHz` positive edge. To address this, we implemented register storage for these values. However, this caused the pixel output based on `h_cnt` and `v_cnt` calculations to experience a clock delay. Therefore, we modified the VGA module to output additional signals: `h_cnt_1`

and `v_cnt_1` representing the `h_cnt` and `v_cnt` values one clock cycle ahead, and similarly, `h_cnt_5` and `v_cnt_5` representing values five clock cycles ahead.

5.2 Layer Implementation

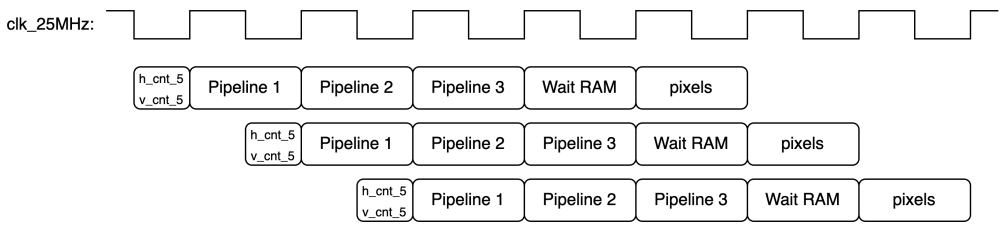
5.2.1 Statis Objects

As shown in Figure 5.2, each Render Module utilizes a chain of if-else statements for output determination. For Static Objects (all objects except the Instances from both sides), implementation is relatively straightforward due to their fixed display ranges and color codes.

Initially, we implemented our design based on intuitive thinking. However, numerous Timing Issues emerged in the Implementation Design Panel in Vivado. To resolve this, we optimized the address calculation process using **pipelining**, computing portions sequentially. The calculated address through pipeline serves as RAM (IPs) `addr` input to retrieve pixel identification codes (converted from PNG to coe codes using Python).

With the pixel identification codes for each object, the entire if-else chain essentially operates on two conditions: whether `h_cnt_1` and `v_cnt_1` fall within the object's rendering range, and whether its pixel identification codes do not represent transparency. When both conditions are met, the if condition is entered, and the pixel ID codes are converted to 12-bit VGA output color codes. If either condition fails, the chain continues to the next if-else statement to check if any other object requires rendering at this pixel. If no objects need rendering, the base layer background is rendered, such as sky, grass, or field background colors, etc.

Due to delays from both pipelining and RAM pixel identification codes' retrieval, calculations showed that we need to use `h_cnt_5` and `v_cnt_5` as reference values at the pipeline's beginning to ensure correct pixel alignment with current `h_cnt` and `v_cnt` during output.



▲ Figure 5.3: Pipeline Technique

For Static Objects, we implemented three pipeline stages to calculate the address value which is then fed to RAM for value retrieval. The calculations include subtracting the object's x and y coordinates from h_cnt_5 and v_cnt_5, dividing these values by a scaling factor (ranging from 2 to 4 depending on the object), multiplying v-related value by the image width, and finally adding these two values together with modulo operation of the specific picture size which is used to prevent giving it out-of-range address. The resulting value represents the index of the color code in the object's coe file that needs to be rendered for that specific pixel at that time point.

5.2.2 Character Instances

When rendering instances, we encountered a challenging issue: to successfully render the intended color code for a specific instance, we needed to calculate the address in the desired image at that time point. The problem arose when multiple instances of the same character appeared on the field. This would generate different addresses requesting color codes simultaneously. Similarly, with three, four, or more identical characters, an equivalent number of image-storing RAMs would be required to calculate the color codes.

Consequently, we reduced our initial plan of 16 instances per side to 8 instances, and implemented **True dual-port RAM** to further halve the storage requirements. Initially, we considered modifying the RAM clock to the system's 100MHz `clk` and alternately feeding addresses while storing retrieved color codes, which would reduce storage requirements by another factor of four. However, after careful calculation, we determined that the current storage capacity was sufficient without implementing this additional measure.

Our implementation consists of four True dual-port RAMs for storing cat images and four for storing enemy images, with each RAM pair responsible for computing color codes for two instances.

The address calculation method for instances differs slightly from Static Objects. We need to calculate a difference value (distinct from `D` in the Pixel Decoder, this offset represents the displacement from the initial point to the starting point of the desired frame based on the current state among each character's six frames). This is accomplished using an additional

decoder, PicNum_By_State, which converts the character's state into the corresponding `picNum` (indicating which of the six frames to use), then multiplying it by `W` and `H` obtained from the Pixel Decoder.

Notably, the PicNum_By_State Decoder takes not only the character's state as input but also the fifth least significant bit of the character's 10-bit x-coordinate. This determines which frame to display during `ST_MOVE`, enabling fluid alternating animation frames as the character moves across the screen based on x-coordinate changes.

Through these calculations, we can obtain pixel ID codes corresponding to each instance at the specific time point. By incorporating these codes along with their x and y coordinates into the if-else chain, similar to Static Objects, we successfully achieve the rendering of character instances.

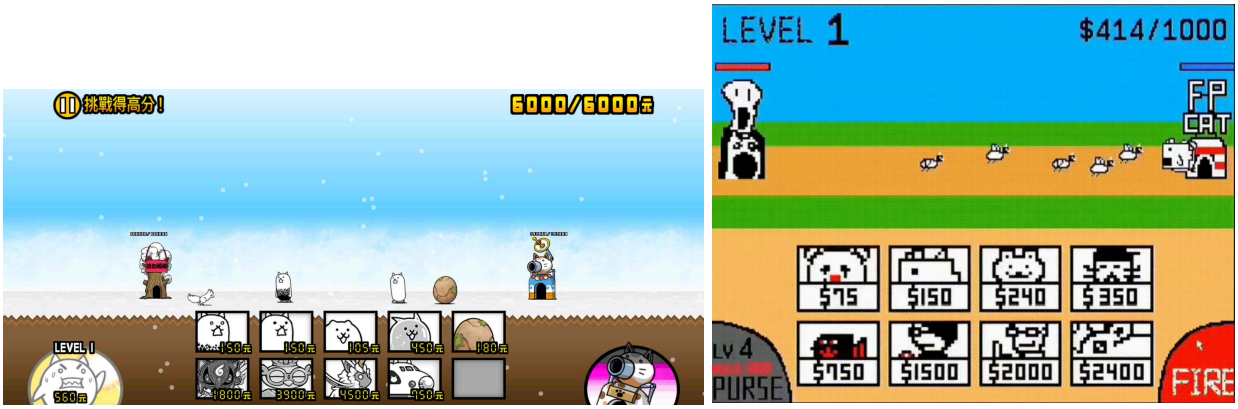
6. Conclusion

6.1 Gameplay Demonstration

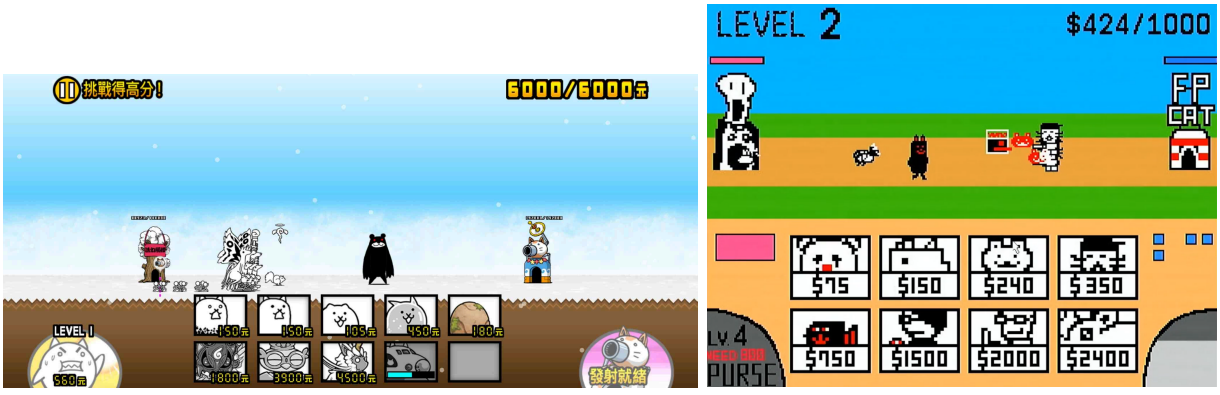
The game's visuals and functionalities have been thoroughly detailed in Chapter 2. Below is a YouTube link demonstrating our gameplay experience across three Stages, showcasing the implemented features of our game, including: maxing out the Purse Level, firing the air cannon, summoning all available cats, toggling the pause switch, and displaying both victory and defeat screens.

YOUTUBE LINK of GAMEPLAY DEMO (TODO)

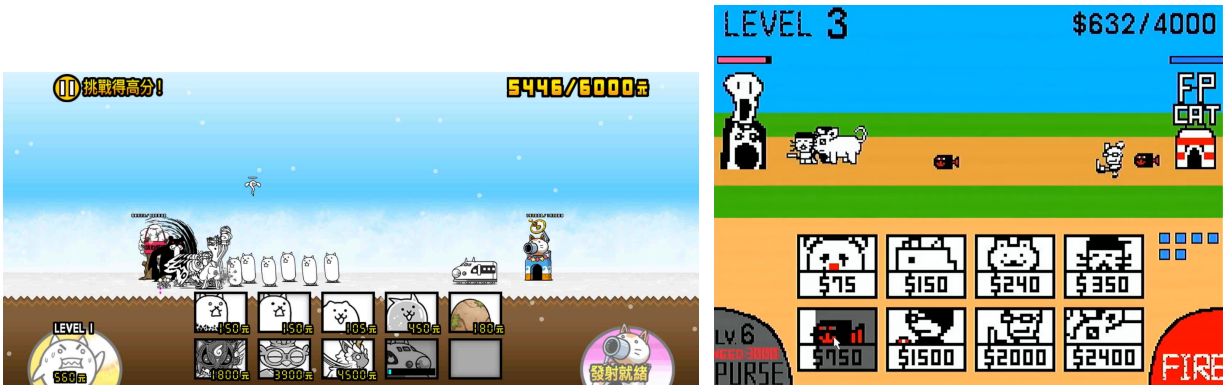
6.2 Comparison with Original Game



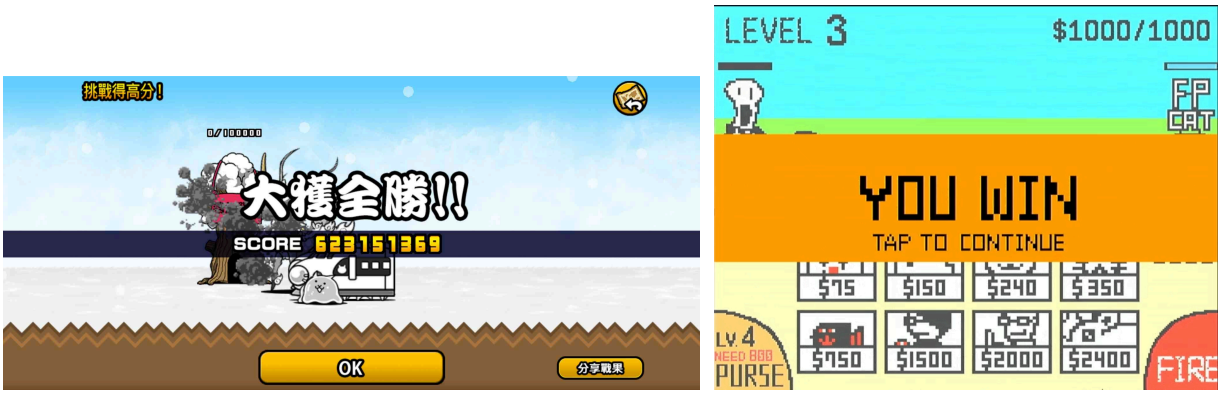
▲ Figure 6.1: Just showing the screen layout of the original game and our game



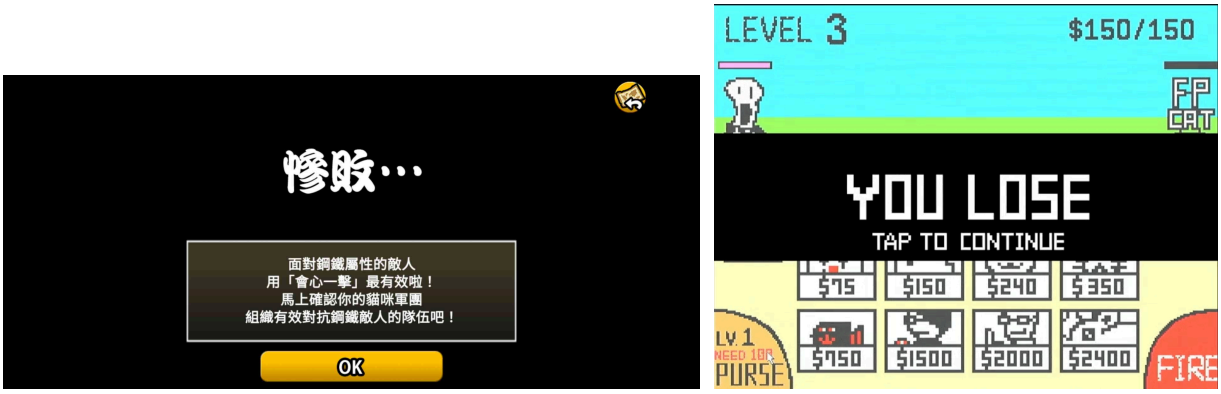
▲ Figure 6.2: Black bear is coming!!! Watch out!!!



▲ Figure 6.3: Don't worry! Bomb Cat (Express Cat) is here.



▲ Figure 6.4: Hurray! We are winning!



▲ Figure 6.5: No…

6.3 What We Learned

After investing over two hundred hours of dedicated effort, we have successfully brought this game to fruition. The journey encompassed every phase - from the initial concept and detailed design considerations to architecting the module structure and establishing necessary protocols. We mastered the configuration of additional memory blocks and developed Python solutions to convert numerous hand-drawn PNG files into COE format. We deliberated extensively on implementing efficient clock utilization to drive the game's operations.

The development process presented significant challenges. When designing individual game states and their interconnected logic, we encountered memory constraints during character rendering that initially left us deeply discouraged and contemplating alternative projects. However, we overcame this obstacle by implementing dual-port RAM and optimizing entity counts. Subsequently, we faced approximately 700 warnings and critical warnings related to time delay issues. Through continuous learning and improvement, including the integration of pipeline design, we successfully resolved these warnings.

This Final Project has been an invaluable learning experience. Beyond enhancing our proficiency in Verilog and Vivado, it cultivated our ability to navigate and overcome new challenges through creative problem-solving. The collaborative nature of the project distinguished it from individual assignments, requiring careful coordination and communication to define module functionalities and establish data storage and transfer protocols. Additionally, we significantly improved our GitHub proficiency.

In conclusion, while this project demanded substantial dedication and effort, the knowledge and experience gained have been immensely rewarding.

A. Appendix

A.1 Python Tool - PNG to COE

To facilitate the conversion between PNG images and COE format, we developed two Python scripts utilizing the **PIL (Python Imaging Library)** package. These tools implement different encoding strategies based on the image type requirements.

The **png-to-coe-1bit.py** tool processes single-color images by analyzing the alpha channel of each pixel. It checks the transparency threshold ($a < 128$) to determine the binary encoding. The script writes the output in the required COE format, including the radix declaration and proper vector formatting with commas and semicolons.

For multi-color images, **png-to-coe-2bit.py** implements a more sophisticated color analysis algorithm. It first checks pixel transparency, then analyzes RGB values to distinguish between red ($R > 200$, $G < 100$, $B < 100$) and grayscale colors. For grayscale determination, it calculates the average of RGB values and applies a threshold of 128 to differentiate between black and white.

Both tools handle proper file extension validation and provide user feedback during the conversion process. The implementation includes error handling for file operations and image processing, ensuring robust operation even with malformed input files.

These Python tools served as essential tools in our development flow, bridging the gap between graphic design and hardware implementation phases of the project.