CHIR_GAB ELECTRONICS
Spring 2023
Anteater Project

COLUMBIA UNIVERSITY

Prepared By:
Chirag Chaturvedi (cc4880) & Gabriela Gonzalez (gng2112)
https://github.com/gab-hub/Anteater

Anteater Logo obtained here

Table of Contents

Section 1: Introduction

Section 1.1: Inspiration

After realizing that one of our teammates, Chirag, had never played pinball *and* we needed to find a game to create for this class, we did some *market research* and went to Barcade to knock two birds with one stone.

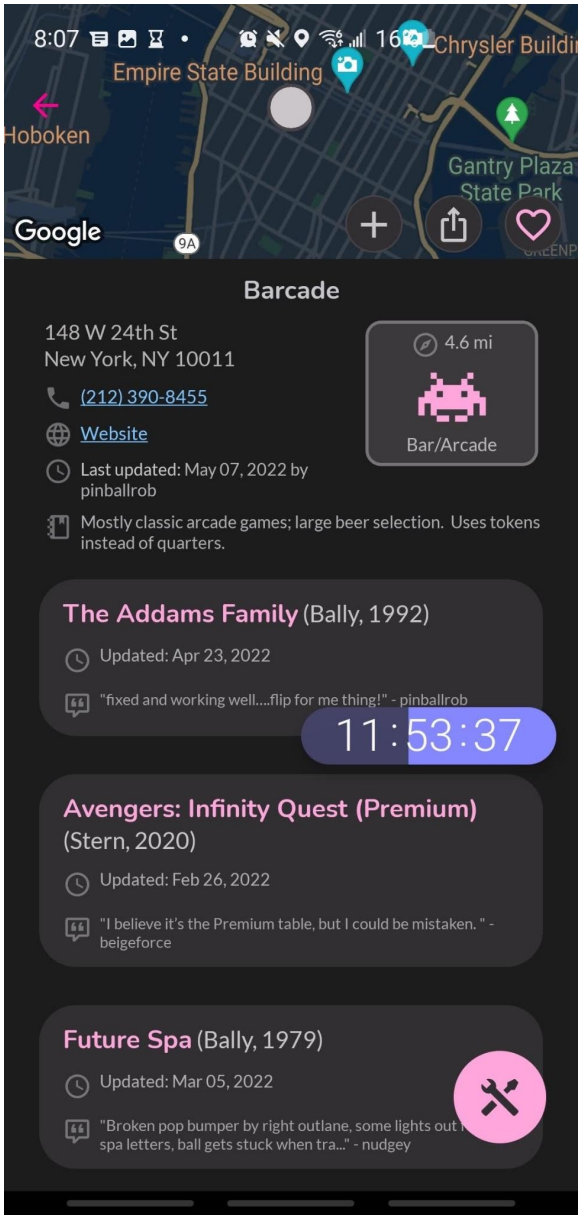


Figure 1: *Initial proposal media for trip*

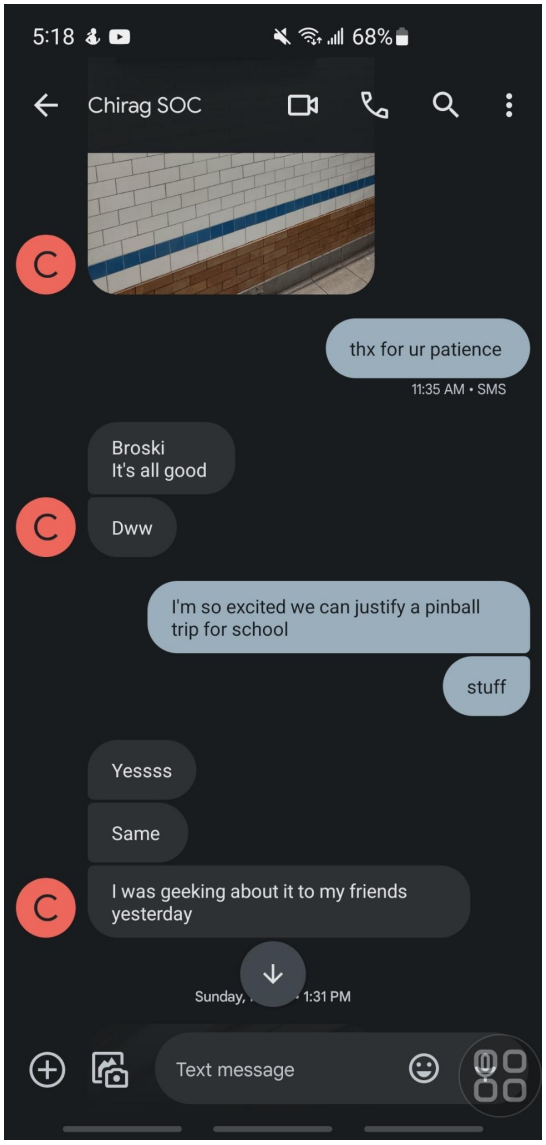


Figure 2: *Two geeks geeking-out*

This is when we saw Anteater, a pacman-like game that really made us feel like horrible arcade gamers. After seeing that not many people had done this game implementation, we decided to take-on the challenge and tackle this game.

Section 1.2: History of Game*

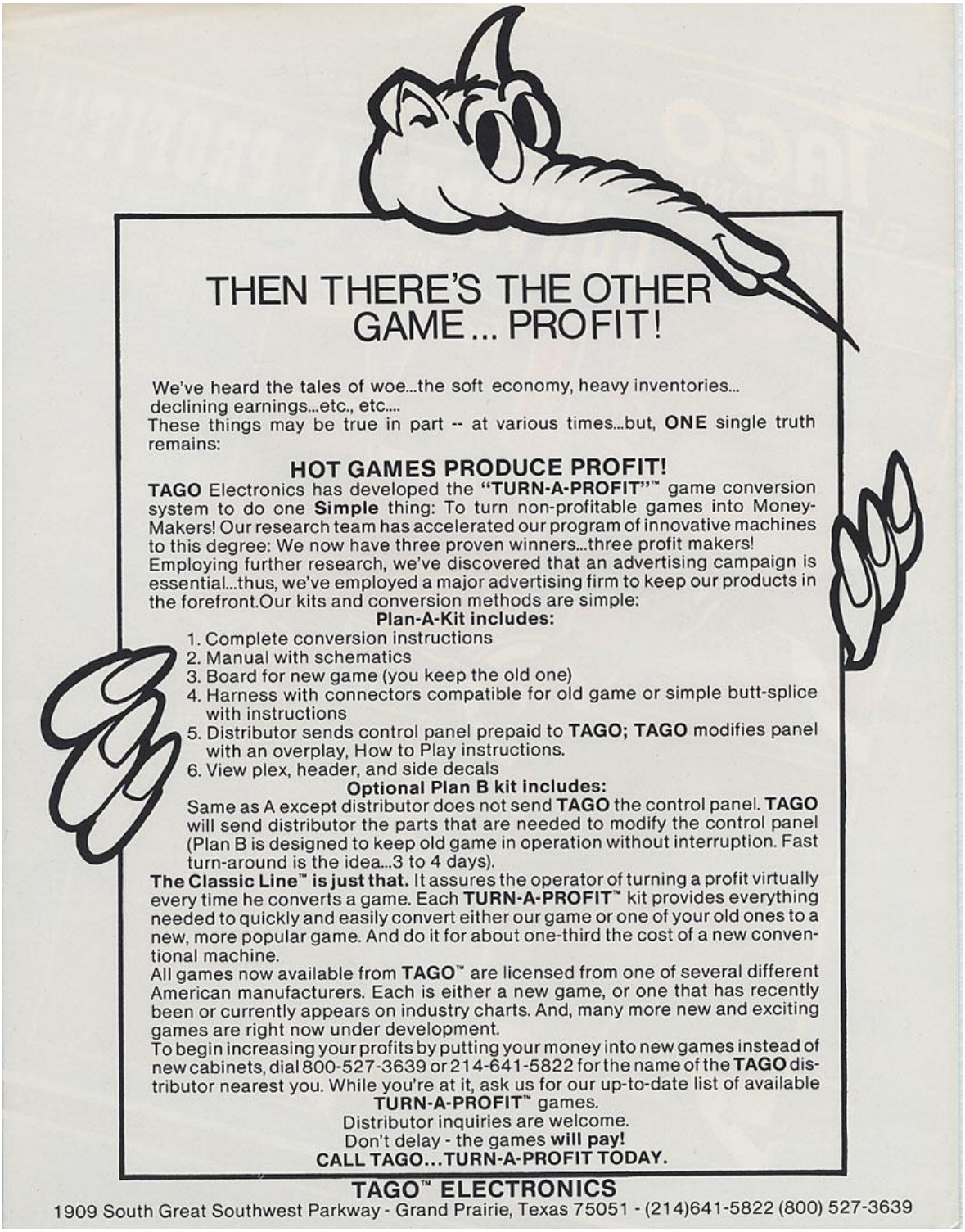
**Note: Due to the limited information about this game, all the information acquired for this section is located on this website*

Originally made by Tago Electronics in 1982, *Anteater* was a part of the only project the company embarked-on. This game was a part of a three game cabinet including *Calipso* and *Video Hustler (Tago)*.

Figure 3: *Anteater cabinet with original side art but a converted Stern cabinet*

This game, according to the Arcade Museum, is classified as an uncommon game.

Figure 4: *Original Tago cabinet flyer*



THEN THERE'S THE OTHER GAME... PROFIT!

We've heard the tales of woe...the soft economy, heavy inventories... declining earnings...etc., etc....
These things may be true in part -- at various times...but, **ONE** single truth remains:

HOT GAMES PRODUCE PROFIT!

TAGO Electronics has developed the **"TURN-A-PROFIT"**™ game conversion system to do one **Simple** thing: To turn non-profitable games into Money-Makers! Our research team has accelerated our program of innovative machines to this degree: We now have three proven winners...three profit makers!
Employing further research, we've discovered that an advertising campaign is essential...thus, we've employed a major advertising firm to keep our products in the forefront.Our kits and conversion methods are simple:

Plan-A-Kit includes:

1. Complete conversion instructions
2. Manual with schematics
3. Board for new game (you keep the old one)
4. Harness with connectors compatible for old game or simple butt-splice with instructions
5. Distributor sends control panel prepaid to **TAGO; TAGO** modifies panel with an overplay, How to Play instructions.
6. View plex, header, and side decals

Optional Plan B kit includes:

Same as A except distributor does not send **TAGO** the control panel. **TAGO** will send distributor the parts that are needed to modify the control panel (Plan B is designed to keep old game in operation without interruption. Fast turn-around is the idea...3 to 4 days).

The Classic Line™ is just that. It assures the operator of turning a profit virtually every time he converts a game. Each **TURN-A-PROFIT**™ kit provides everything needed to quickly and easily convert either our game or one of your old ones to a new, more popular game. And do it for about one-third the cost of a new conventional machine.
All games now available from **TAGO**™ are licensed from one of several different American manufacturers. Each is either a new game, or one that has recently been or currently appears on industry charts. And, many more new and exciting games are right now under development.
To begin increasing your profits by putting your money into new games instead of new cabinets, dial 800-527-3639 or 214-641-5822 for the name of the **TAGO** distributor nearest you. While you're at it, ask us for our up-to-date list of available **TURN-A-PROFIT**™ games.

Distributor inquiries are welcome.
Don't delay - the games **will pay!**
CALL TAGO...TURN-A-PROFIT TODAY.

TAGO™ ELECTRONICS
1909 South Great Southwest Parkway - Grand Prairie, Texas 75051 - (214)641-5822 (800) 527-3639

Figure 5: *Original Tago Electronics advertisement flyer*

Section 1.3: How to Play *Anteater*

To see a professional play the game, see this link.

Here is the list of the basic original game-play:

- Capture as many ant larvae with the tip of the anteater tongue, similar to Pac-Man
- Capturing certain animals add more points to your score
    - Ants can be captured from any position in their body, but the tongue trail cannot hit an ant's path. Otherwise, it's game over.
    - Worms can only be captured from behind. Otherwise, it's game over.
    - Spiders come-out at night and will attempt to traverse the trail of the tongue. If the spider reaches the tip of the tongue, it's game over.
    - Queen ants on the bottom are stagnant and eliminate all the animals on the screen to allow the user to capture ant larvae.
- The level is completed when all the ant larvae are captured.
- Players can traditionally utilize a joystick to traverse through the maze.
- If players sense danger, they can retract the tongue by pressing a button traditionally located next to the joystick.

Section 2: System Architecture

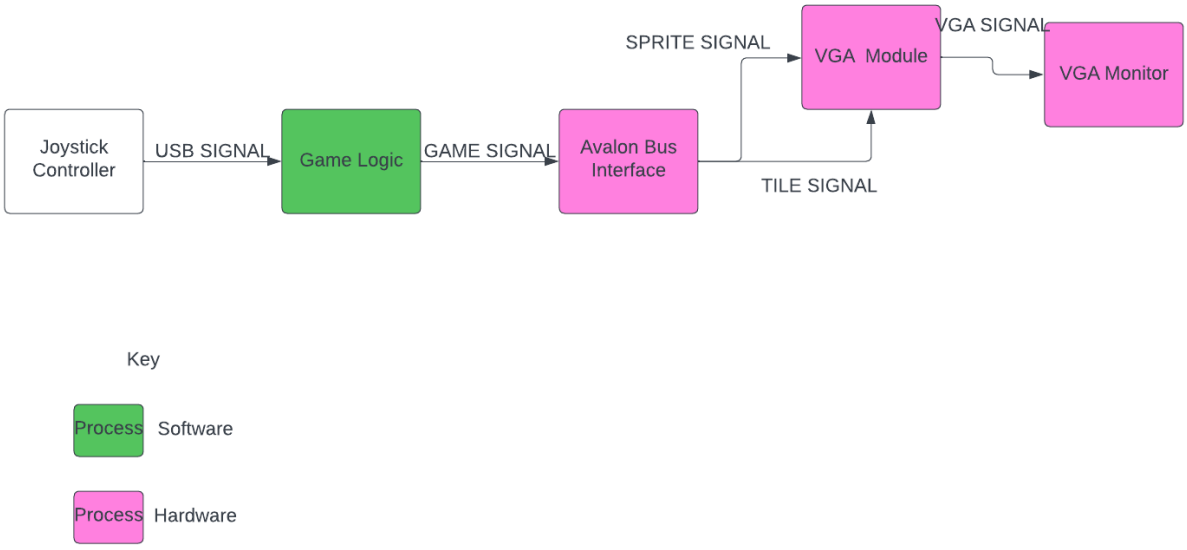


Figure 6: *System architecture for Anteater Project*

The system architecture for the game consists of a Joystick controller that sends control data from the user to the software. The software controls the game logic and, depending on game conditions, sends the appropriate signal to the Avalon bus interface. At each cycle, 4-bits of data are read at a time to display background tiles, sprites, and other game visuals. The hardware code is stored in a single System Verilog file but can be split into two modules based on function. The displays the background from the ROM modules and the sprites from the RAM module utilizing the on-chip RAM and data transmitted from the software. Further details of the design will be explained in the hardware section.

Section 3: Hardware

Section 3.1 Graphic Design

Modifications were made to the vga_ball.sv file from lab3 in order to implement our game.

Section 3.1.1 Maze Tiles

		Design Decisions
Storage Location	ROM module	Unchanging data
Memory size	32 x 32 bits	Easy storage on FPGA
Quantity	10	Repeated layout because of limited memory
Total background size	128x52 bits	Centered on screen to follow original game dimensions

Hardware design implementation	Hardcoded Bitmap
	`bgndD2_c[0] = 32'b0000_0000_0000_0000_0000_0011_0011_0011;`
	`bgndD2_c[1] = 32'b0000_0000_0000_0000_0011_0100_0100_0011;`
	`bgndD2_c[2] = 32'b0000_0000_0000_0011_0100_0100_0011_0100;`
	`bgndD2_c[3] = 32'b0000_0000_0011_0011_0101_0101_0011_0100;`
	`bgndD2_c[4] = 32'b0000_0011_0101_0011_0101_0101_0011_0100;`
	`bgndD2_c[5] = 32'b0011_0011_0011_0011_0011_0100_0100_0011;`
	`bgndD2_c[6] = 32'b0011_0100_0101_0011_0100_0100_0011_0011;`
	`bgndD2_c[7] = 32'b0011_0100_0100_0011_0011_0100_0100_0011;`
		Implement palette of colors

Table 1: *Maze tile information*

Background tiles are stored in a ROM module because this is data that is not changing; it is the underlying layer in which sprites will go on top of. Bits are read 4 bits at a time in order to assign specific pixels to colors on our palette.

The following flip flop logic assigns them to the positions on the VGA board given the overall bitmap of the whole game layout. In the bitmap for the whole game, each tile (and sprite) is referencing the numbers in the flip flop cases.

```
always_ff@(posedge clk) begin

  //if(readreq) //if there is a readrequest output data
    case(tile)
      4'd0:begin
           data <= sky[addrY];
           end

      4'd1:begin
           data <= bgndA1_c[addrY];
           end

      4'd2:begin
           data <= grass[addrY];
           end

      default: data <= 4'b0110;
    endcase

end
```

Figure 7: *Tile background logic*

```
bmap1[0]  = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[1]  = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[2]  = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[3]  = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[4]  = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[5]  = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[6]  = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[7]  = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[8]  = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[9]  = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[10] = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[11] = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[12] = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[13] = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[14] = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[15] = 128'b0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010_0010;
bmap1[16] = 128'b0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0000_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001;
bmap1[17] = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[18] = 128'b0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0000_0001_0001_0001_0001_0001_0001_0000_0001_0001_0001_0001_0001_0001_0001_0001;
bmap1[19] = 128'b0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0000_0001_0001_0001_0001_0001_0001_0000_0001_0001_0001_0001_0001_0001_0001_0001;
bmap1[20] = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[21] = 128'b0001_0001_0001_0001_0001_0000_0000_0001_0001_0001_0001_0001_0001_0001_0000_0001_0001_0001_0000_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001;
bmap1[22] = 128'b0001_0001_0001_0001_0001_0000_0000_0001_0001_0001_0001_0001_0001_0001_0000_0001_0001_0001_0000_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001;
bmap1[23] = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[24] = 128'b0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0000_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001;
bmap1[25] = 128'b0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0000_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001;
bmap1[26] = 128'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
bmap1[27] = 128'b0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0000_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001;
bmap1[28] = 128'b0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0000_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001_0001;
```

Figure 8: *Early iteration of game tile bitmap*

Section 3.1.2 Other Tiles

Type	**Size (bits)**
Grass	32x32
Sky	32x32

Table 2: *Other tile information*

Section 3.1.2 Sprites

		Design Decisions
Storage Location	RAM module	Animations, not smooth movements
Memory size	Varies	Dependent on size wanted on screen
Quantity	6	Capture different movements on sprites

Table 3: General *sprites information*

Type	**Size (bits)**
Anteater	128x8
Sun	68x17
Ant	32x32
Worm	32x32
Larvae	32x32

Table 4: *Specific sprite information*

Section 3.2 Color Pallette

The 4 bit binary numbers in the sprite and tile locations correspond to the following colors. By doing the mapping in this way, we can save memory by just having a look-up table of hex values.

Bit number	**Color**	**Hex Representation {VGA_R, VGA_G, VGA_B}**
0000	Navy blue (background)	{8'h00, 8'h0c, 8'h66}
0001	Orange (anteater)	{8'hff, 8'h5e, 8'h0e}
0010	Aquamarine (score/worm)	{8'h75, 8'he6, 8'hda}
0011	Tan (maze)	{8'hff, 8'h8c, 8'h00}
0100	Magenta (tongue)	{8'hff, 8'h00, 8'hff}
0101	Green (grass)	{8'h3c, 8'hb0, 8'h43}
0110	Black (details)	{8'h00, 8'h00, 8'h00}
0111	White (details)	{8'hff, 8'hff, 8'hff}
1000	Electric blue (worm)	{8'h00, 8'h33, 8'h99}
1001	Red (sun)	{8'hff, 8'h00, 8'h00}

Table 5: *Specific sprite information*

Section 3.2 Handling Delays

Because we are doing a hardware implementation and considering the timing of the clock cycle, in order for our tiles and sprites to be positioned in the positions we intended, we need to have variables to load certain information early like xpos. Otherwise, we get tiles that are not aligned correctly.

```
always_ff @(posedge clk) begin
    {VGA_R, VGA_G, VGA_B} <= {8'h0, 8'h0, 8'h0};
    if (VGA_BLANK_n ) begin
// starting position                                    ending position
if (hcount[10:1] >= 10'h96 && vcount[9:0] >= 10'h1 && hcount[10:1] <= 10'h220 && vcount[9:0] <= 10'h114) begin
    xpos <= (hcount[10:1] - 10'h96) >> 3;
    yp <= (vcount[9:0] - 10'h1) >> 3;
    addrX0early <= (xpos << 3) + 10'h96;
     addrY <= vcount[9:0] - ((yp << 3) + 10'h1);
    addrX0 <= (hcount[10:1]-addrX0early - 1);
```

Figure 9: *Flip flop with logic to obtain correct game bitmap position*

Section 3.3 Final Qsys Connections

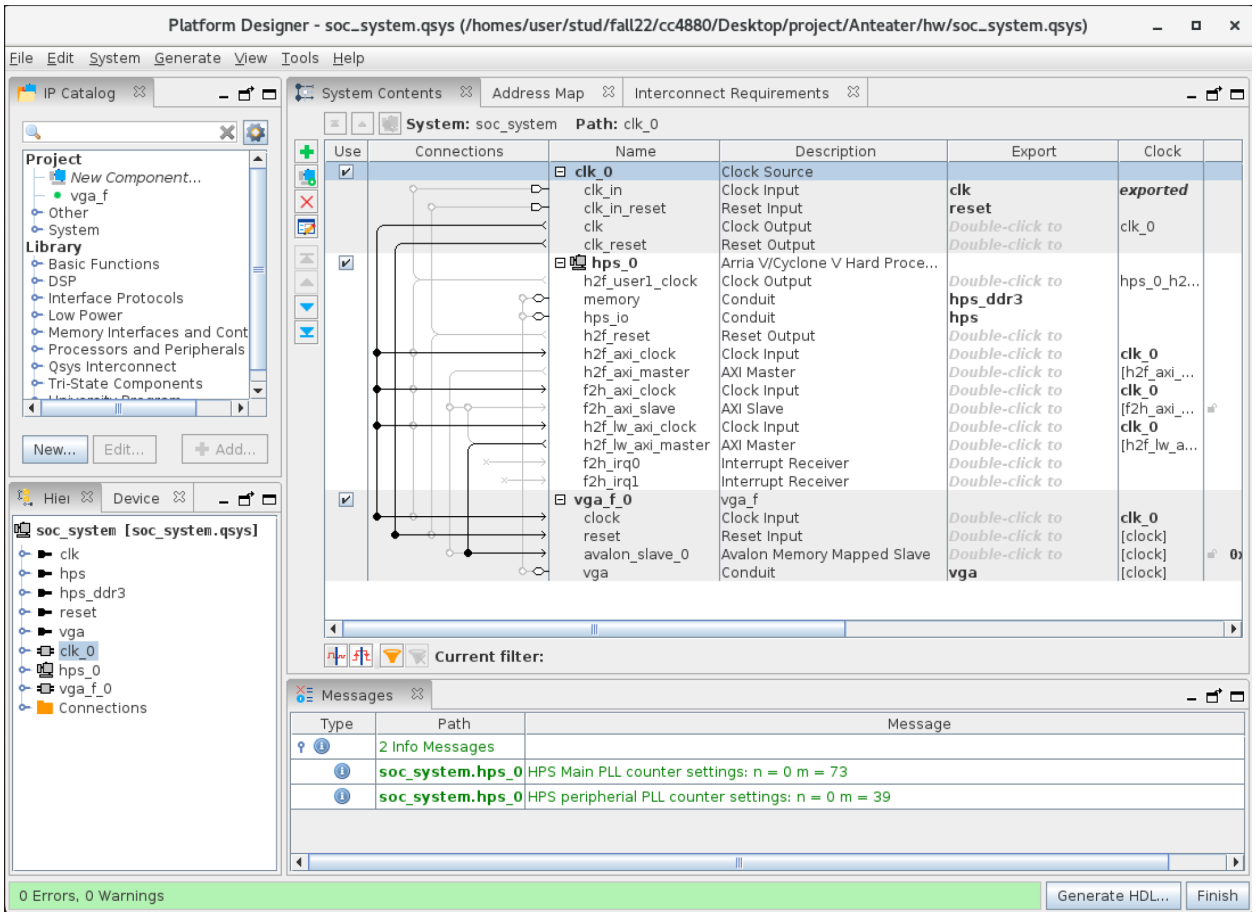


Figure 10: *Final Qsys connections*

Section 4: Software

Section 4.1: Joystick

We implemented our project with the NES USB controllers as depicted below.

Figure 11: *NES USB controller with how they were mapped to be utilized*

Section 4.2: Game Logic

Creature	**Score Addition**
Larvae	+10
Ant	+100
Worms	+200

Table 6: *Scoring Table*

Section 4.3: Data flow

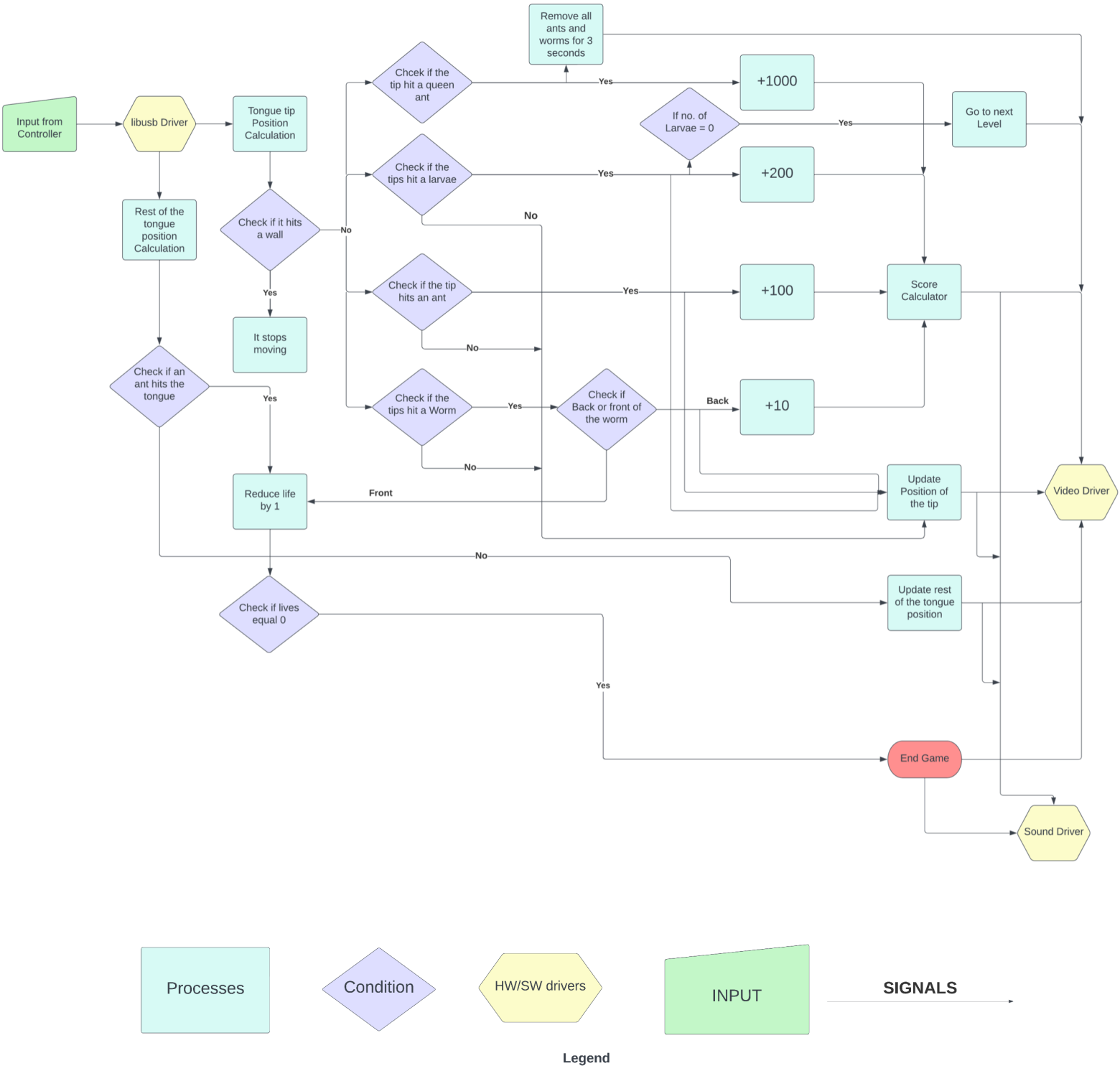


Figure 12: *Data flow diagram of the Anteater game*

Section 4.4: Process Details

Input from the controller: Users can control the horizontal and vertical position of the tip of the tongue with the help of the joystick on the controller. They can also retract the tongue with the help of a button on the controller. We will be using an Xbox 360 controller which connects to the PC with a USB connector, this enables us to use the library “libusb” for getting signals from the controller.

Tongue Tip Position Calculation: After the input has been received, the position of the tip will be calculated, this is going to be pretty simple as we don’t have to consider how much the joystick has been pushed and all we have to worry about is the angle as the tongue will only be moving either vertically or horizontally. During this process, we will be sending the updated position continuously to the video driver. If the next position is the wall and not a free path, the tongue will stop moving and it will wait for the next input.

Tongue Position Calculation: The tip of the tongue and the tongue are 2 separate entities in the game as they interact with other objects in the game very differently from each other, as the tip moves it leaves a trail of the tongue behind it, and we will have to keep track of all the positions it is on as collision of the tongue with ants or worms can lead to different outcomes, the user needs to keep track of the tongue as well. During this process as well we will have to constantly send position data to the video driver so that it can be reflected on the screen.

Collision of the tip with Ants, Worms, and Larvae: If during movement the tip collides with an Ant then the Ant disappears and we add 100 points to the score, this data is immediately sent to the video driver, if it collides with a worm then we check whether if it has collided with the front of the worm or the back If its the front, then we reduce the lives by 1 and this data is sent to the video driver to show on the screen, else if the tongue hits the back of the worm then the worm disappears and we add 200 points to the score, this data is immediately sent to the video driver and is reflected on the screen. If the tongue hits larvae, the larvae are made to disappear and a score of +10 is added, if it is the last larvae present in the game, then we progress to the next level, this data is also sent to the video driver immediately.

Collision of the tongue with Ants and Worms: If during movement an Ant collides with the tongue, then the number of lives is reduced by 1, if a worm collides with a tongue then it just passes through, this data is immediately sent to the video driver so that It can be reflected on the screen, along with that we also check if the no. of lives = 0, because if it does then the game ends there and this data is also immediately sent to to the video driver.

Section 4.5: Object-Oriented Programming

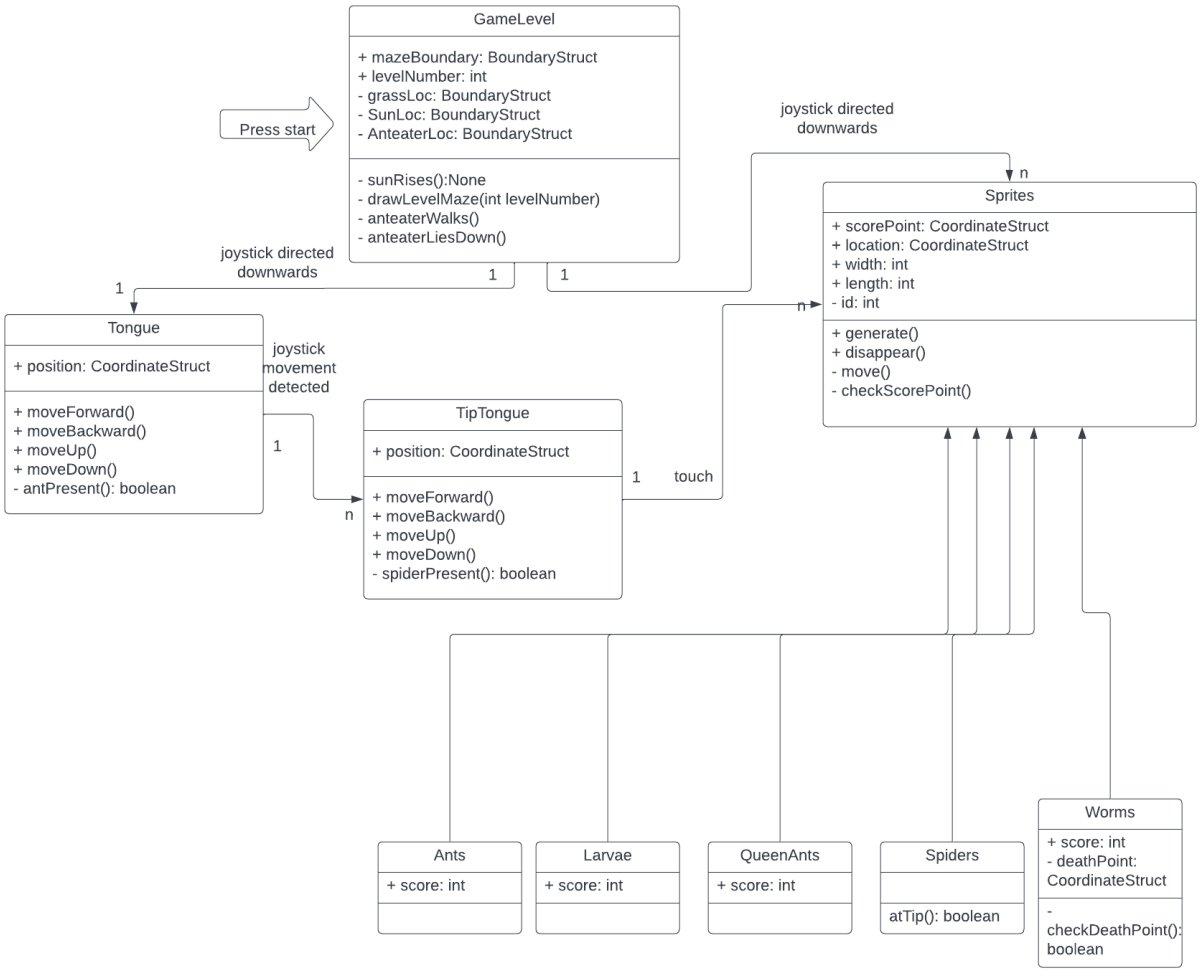


Figure 13: *UML Diagram of the Anteater game*

The GameLevel class will create the scaffolding for the level, like setting the maze boundaries, that is contingent on the degree of difficulty for the level. Additionally, grass will be placed on top of the maze to emulate the original game graphics. For the maze and grass, there will be a BoundaryStruct created to indicate the exact position of the negative space in the maze.

Once the maze has been set, the anteater can walk out and place itself at the opening to prepare for gameplay. In the midst of this, the sun will rise or set depending on the level count. Depending on the level, the anteater will be standing on its legs or lying down, so we added a function to take this into account.

When the joystick is moved, the tongue will move and its position will be stored so that we can display the path the tongue has taken. Additionally, the tongue position will determine some end-game conditions that will be specified in the Sprites class. The tongue will have functions that will be triggered by the joystick to move forward, backward, up, and down. The Tongue class will also have methods such as antPresent(), an endgame condition.

Following the tongue will be the TipTongue to denote the attributes for the tip of the anteater tongue. Only the tip of the tongue can touch the sprites to acquire points. It will have the same attributes as the tongue class class. Additionally, spiderPresent() method will be utilized to check if an ant has hit the tongue or if a spider has reached the tip of the tongue, an endgame condition.

The Sprites parent class will hold information about its location, width, and length, id (identifier). The attribute scorePoint corresponds to the spot on the sprite that the tip must touch in order to score a point and cause the sprite to disappear. The functions generate(), disappear(), move() will change the location of the sprite while the checkScorePoint() will check if a sprite has been “eaten.”

The Ants, Larvae, and Worms classes will have an attribute score to indicate how much they are worth. The Spiders class will just check if a spider is at the tip of the tongue.The Worms class will have a score similar to the other sprites, but it will also have a deathPoint attribute. Since a worm can only be eaten from behind, deathPoint indicates the point the tip needs to hit in order to lose. There will be a checkDeathPoint() method to check if the death point has been touched.

Section 5: Process, Process, Process

With only two people in the group, it was vital to have a process to ensure this large project would get completed. Here are some of the things we did to ensure success.

Section 5.1: Partner Programming

During the labs and during the project, especially when we were getting familiar with SystemVerilog, it was vital for us to sit together and talk through our thought process and explain code to each other. This allowed for both of us to learn while fact-checking our logic.

Figure 14: *Timing diagram utilized during partner programming*

Section 5.2: Git branching

During the implementation of different features, it was vital to have different features on different branches and only merge to the main branch when code worked.

Section 5.3: Simulate a small map in software

Before the hardware and software integration, we made a smaller map in order to catch end-cases quicker without dealing with the resource overload of traversing a larger maze. After all the bugs were smoothed-out of the logic, it was modified for our actual maze.

Section 5.4: Sprite and Tile Design

It was really helpful to print on paper pictures of the game and plan out the bit maps on paper to better visualize the different movements.

Additionally, in order to capture the movements of the animations, it was helpful to watch YouTube videos of people playing *Anteater*, especially on 0.25 speed.

Section 5.5: AI to Stress Test

Because of the AI implementation one of our teammates, Chirag was doing for one of his other classes, we were able to use the model to stress test the model for us. This allowed for the quicker catching of bugs versus us playing it hundreds of times.

Section 5.6: Small VGA screen

While getting to know the nuances of our SystemVerilog, it was helpful to have a scaled-down version of our game on the VGA monitor for learning purposes.

Section 6: Roles & Advice for Future Groups

Section 6.1: Roles

- Chirag: Hardware design, iterate on software design & stress test, integration
- Gabriela: Initial Software logic, sprite and tile creation, joystick driver adaptation, write report, create slides, integration

Section 6.2: Advice for future groups

- Start early
- You will be spending a lot of time in the lab, almost the whole semester to be precise. Therefore, it is important to make it a habit to go to the lab, even if it's just for a small sprint. This will allow you to not feel rushed in the assignment and give you room for curiosity, where the true learning lies.

Section 7: Code files

Section 7.1: File structures

- hw
    - ip
        - intr_capturer
            - intr_capturer_hw.tcl
            - intr_capturer.v
    - Makefile
    - vga_f.sv
    - Soc_system_top.sv
    - soc_system.qsys
    - soc_system.tcl
- sw
    - anteater_sw.cpp
    - anteater_sw.h
    - joystick.c
    - joystick.h
- other
    - format.py // for formatting sprites
- README.me
- Background_tiles // bitmaps for background tiles
- Sprites // bitmaps for sprites

Section 7.2: Code Appendix

The most up-to-date versions of all the code can be found here.