ECE 385

Spring 2023

Final Project Report

Max Gendeh, Amaan Rehman Shah

HZ/Hongshuo Zhang

5/2/2023

GitHub Link: https://github.com/AmaanRehman/ECE385_FinalProj

Project Journey/Timeline

Our initial project idea was to make a fruit snake collecting game, where a snake starts off tiny and gets larger as it eats more fruits. We wanted to build on this game by implementing multiplayer support, audio, obstacles, and different modes. This would include a main screen in which players can choose their difficulty, mode, audio, etc.

We chose this idea because this game's motion control, which we heard to be the hardest part of most projects, was closely related to Unfortunately, we came to realize that this idea would be extremely difficult to implement just because of the snake body's physics. Specifically, getting the body to trail after the head and enlarge after eating a fruit. After days of heavy photoshop, implementation, and logic testing with no success, we realized there was no way we'd get this working before the demo, and successfully got an extension.

Instead of completely scrapping our idea, we decided to take what we had successfully working, and turn it into another "interesting" concept, a 2-player shooter game. This game had the same WASD motion logic as our previous idea and the same sprites. We called this game "Cobra Combat".

The objective of this game was no longer competing for the most fruits, but a PVP shootout. Since we still had the fruit sprites from our first implementation, we decided to use fruits to refill each player's venom, the projectile the snakes shoot at each other. Fortunately, we were at least able to demo both of our snake's head movement and collision with obstacles to our TA.

The next week was rather slow, due to our lack of knowledge with random generation. We ended up doing this with inputs from the user. In addition, throughout all this, our collision logic was failing for certain cases. Thanks to Ian's help, we figured out that we had the center of our snake's sprite right at the center, instead of the top left, which was how most people were doing. Changing this fixed most of the collision errors, but created other errors with our venom projectiles (for some reason) and code that relied on the position of our snake. Fixing all of this took an extremely long time.

With a few days left, our main focus was getting everything connected. First, we created our home page, and made sure our map selection worked (using a joystick to select between Map 1 and Map 2) and the start button [ENTER] would take players straight to the arena. Most of our time this week, again, was spent making sure random generation of the fruits worked with the selected map of our player's choosing.

With two days left, we got everything working properly. Feeling a bit relieved and experimental, we attempted to add more to our project. The first of these was a third map, using photoshop. However, we noticed as we hovered around 85% of memory usage, the FPGA compilation time slowed down drastically, so we gave up on that. The second thing we tried was a little bit of snake animation, where the snake's tongue would continuously flick its tongue as it roamed around the arena. Unfortunately, this took a lot more effort than we thought with

continuously changing sprites that relied on the positioning of the snake, so we shelved that idea for the time being.

Then, with two days left, we implemented some simpler features to the game, such as snake heart count, projectile count, and snake death animation. In addition, on the morning before the demo, in desperation, we added a funny, but useless easter egg. After around 30 seconds, an easter egg will pop up that a player can consume. When that occurs, Amaan's 24 x 32 sprite will appear on screen for a bit. Finally, after two and a half weeks of camping in office hours, we finished our 2-player Cobra Combat shooter game. This was a chaotic, stressful, but fun experience. I'm extremely grateful to our CAs, who spent extra time out of their hectic day to help us catch that missing bracket in our code or understand why our snake has a seizure every time it collides with a corner.

Introduction

Our final project will be a 2 player snake shooter game, where the objective is to hit the other player with venom shooting out of each snake's head. The player must stay within the game arena, or they lose. Players must engage in combat while avoiding collision with obstacles or shooting them down. A player controls the snake's movement by controlling the direction of the head as it moves around. The snake's body will trail behind the areas on the arena where the head has been on. We will build on this concept by a home/settings screen, animations, and other creative features. Our design will use an on-chip memory to instantiate our background grid for the maze and game arena boundaries.

We designed our top level in SystemVerilog similarly to what we had in Lab 6.2. We have X finite state machine modules, used Ian's helper tool, and C for game logic we had from Lab 6.

A large part of our project's aesthetics was from photoshop (layers shown below). Thanks to the helper tools, we were able to smoothly create and integrate our homemade sprites into our game.

Block Diagram

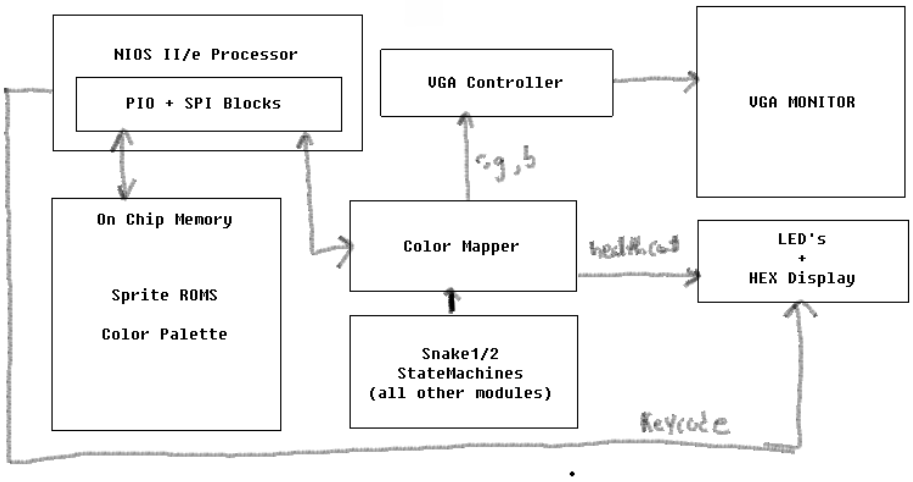


Figure 1: Simplified Block Diagram

Designing of Game Art

As mentioned in the “Project Journey”, we use photoshop for all of our sprites, backgrounds, and obstacle courses. We chose to use Adobe Photoshop for this due to it being very beginner friendly.

To start, we wanted an appealing and colorful background that compliments the color of the green snake. We went with a cyan and turquoise checkerboard pattern as shown below:

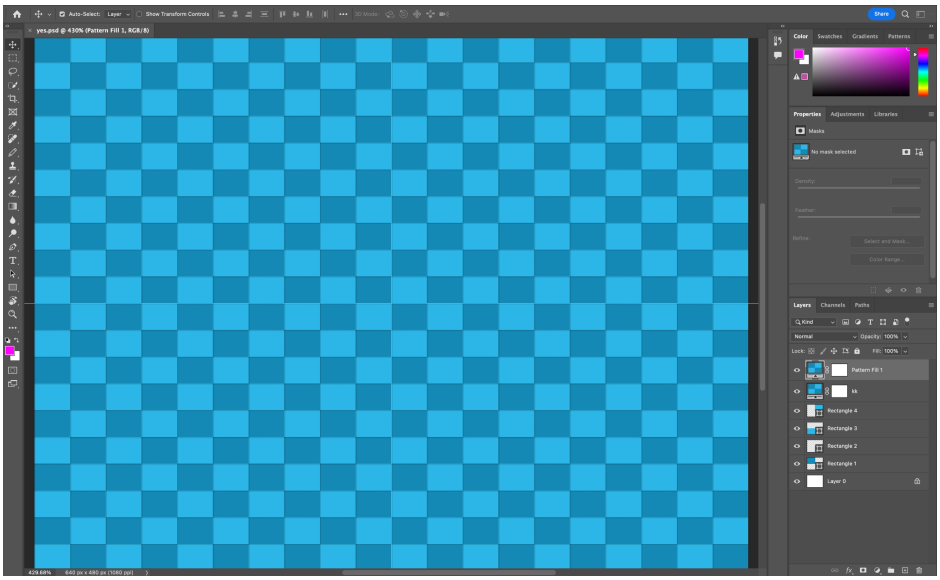


Figure 2: Map 1 Background

Our snakes are also shown below. The reason we made our backgrounds pink was because it had an easy ASCII color h’ FF00FF, which we could easily coded to remove the pink such that only the snake was showing up on the arena.

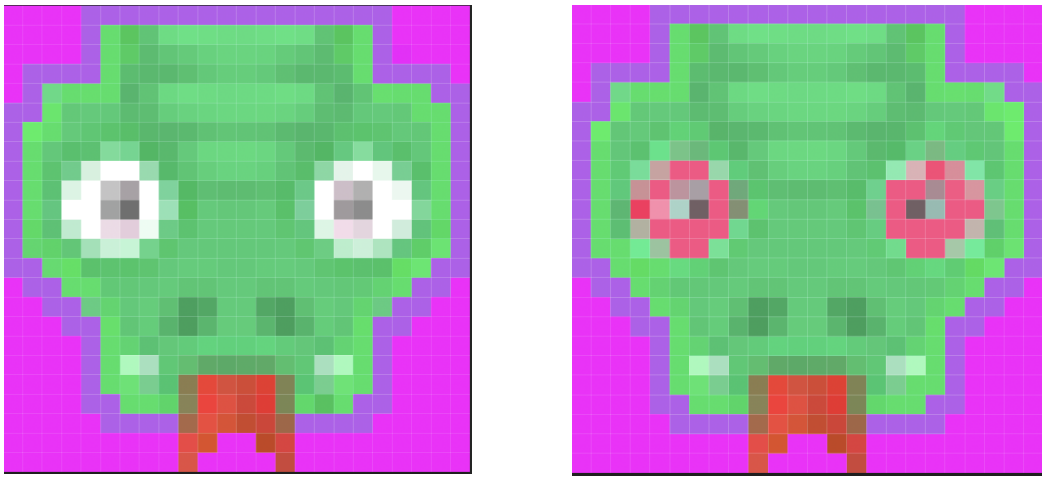


Figure 3: Snake Sprites

To add our obstacle course, we also used a pink background such that we can easily put it on top of our background.

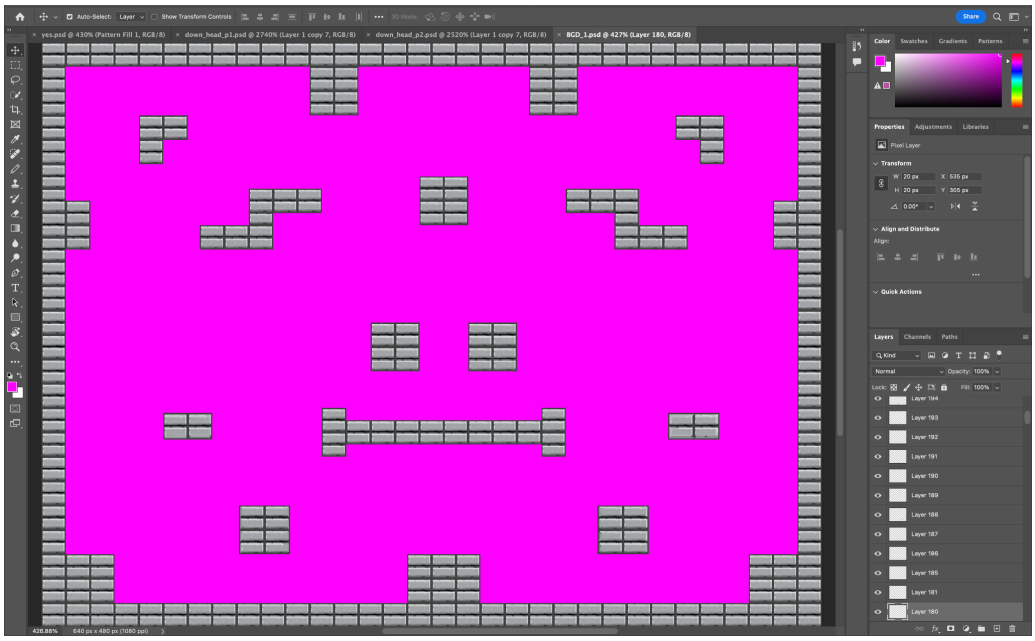


Figure 4: Map 1 Obstacle Course

This was made by using many layers and manually placing them in the shape above. The product we demoed look something like this:

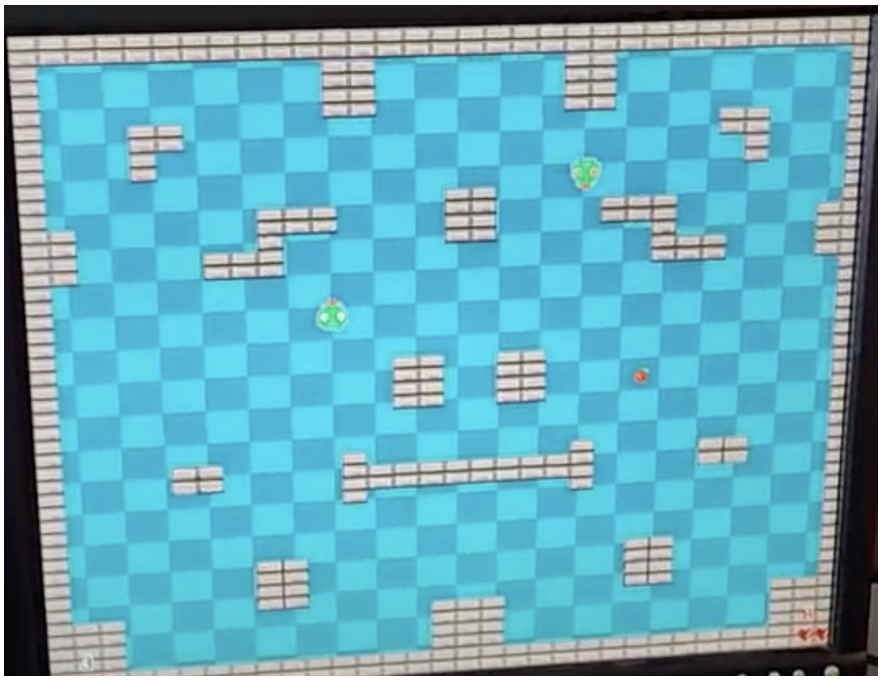


Figure 5: Demo Product

However, that's not all. We had a main menu that we could have used to choose between the two maps using the FPGA joystick and selecting by pressing [ENTER].

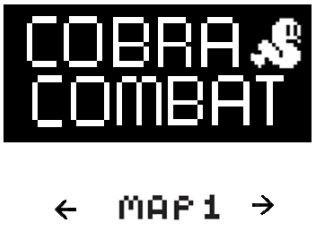
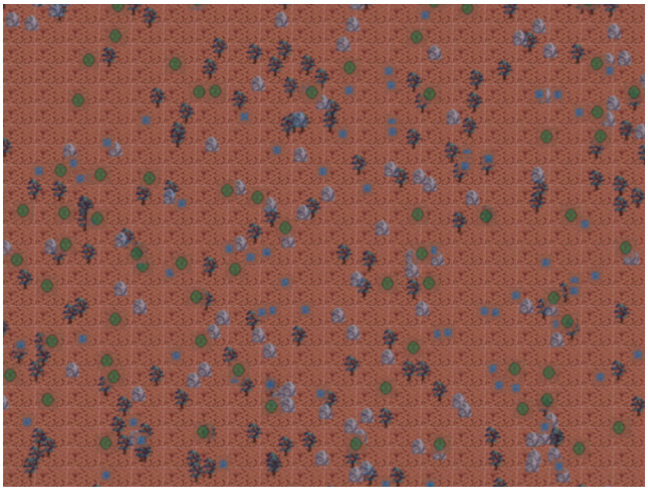
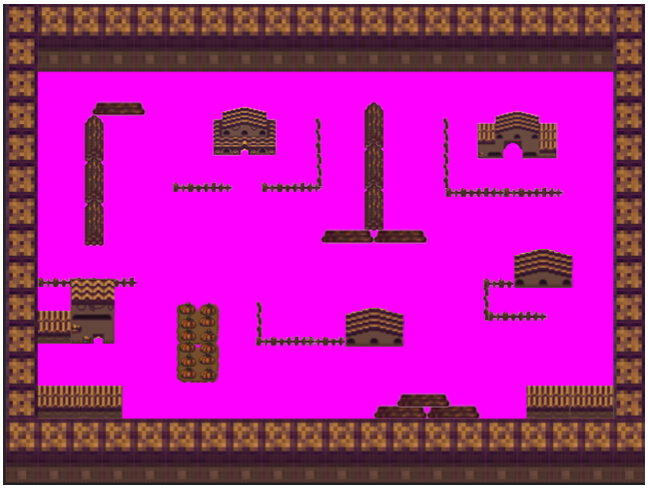


Figure 6: Main Menu

Our second map looked like:





Finally, our end game screen looked like:

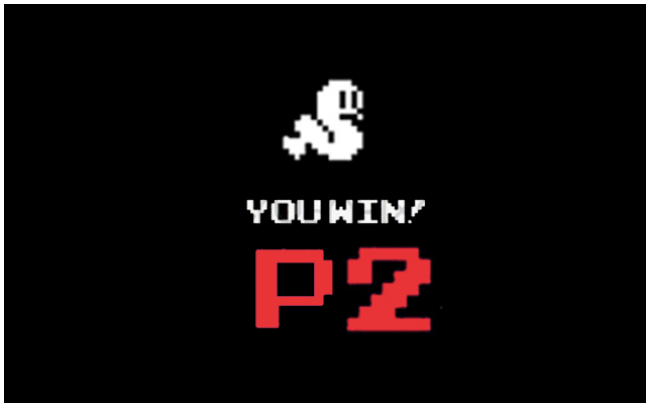


Figure 7: Game Over Screen Variations

Color Mapper RTL View

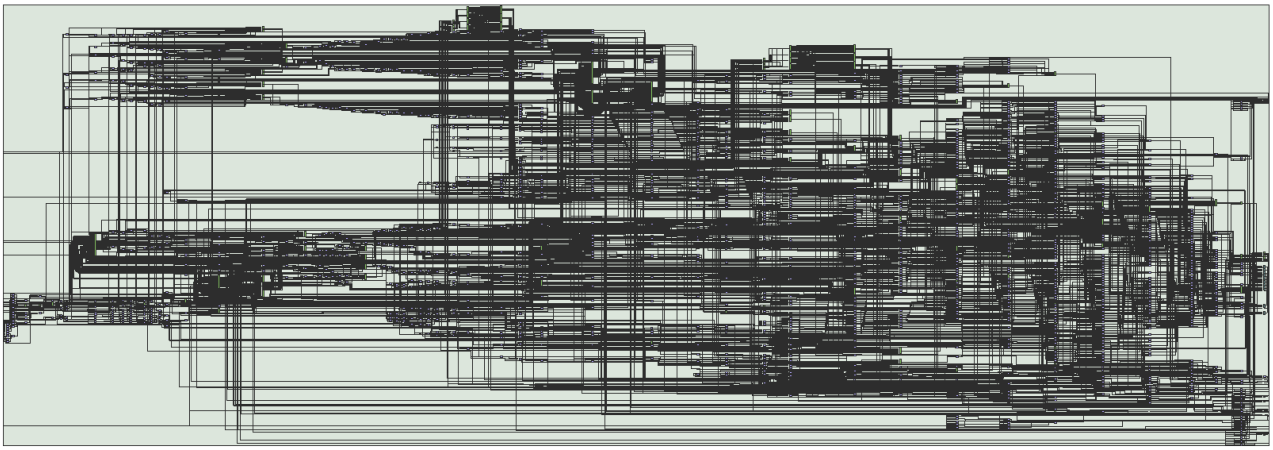


Figure 8: Color Mapper Viewer

Instead of having most of our color logic in our top level, we controlled everything in the Color Mapper (we named ColorBGD). This includes the state machine for detecting user key presses and outputting the appropriate sprite. It also establishes connections with the ROM files for heart and venom sprites. This explains why our color mapper looks extremely condensed while our top level diagram is tame.

Important Module Descriptions

Modules from Lab 6.2/7 used in this final project: Color_Mapper.sv, ball.sv/ball.sv2 (turned into snake.sv), VGA_Controller.sv, font_rom.sv, reg_file.sv, reg_unit.sv (lab 5), ISDU.sv, HexDriver, lab62.soc(turned into finalproj.soc)

- small changes we made ball.sv
    - instead of having 8 bit keycodes, we used 16 since we needed to account for 2 player presses
- Color Mapper is no longer used for foreground or background since we created our own, but for sprites
    - our main ColorMapper is colorBGD_example
- Added our main state machine logic for navigation, animation, snake win condition, and home screen
- Hex Driver is now used to show keycodes and lives of the snake
- Changed lab62.soc's keycodes from 8 bits to 16 bits

Module: ISDU.sv
Inputs: Clk, Reset, [15:0] keycode, player1 wins, player2wins, tie
Output: LD_MENU, LD_Map1, LD_S1ENDGAME, LD_S2ENDGAME, Pause_En, AnimationActivem [2:0[offset, AnimationActive1, [2:0] offset1
Description: Instantiates state machine used to determine end game logic
Purpose: Will bring up the appropriate end game screen when an end game condition is met and the module jumps to an end state

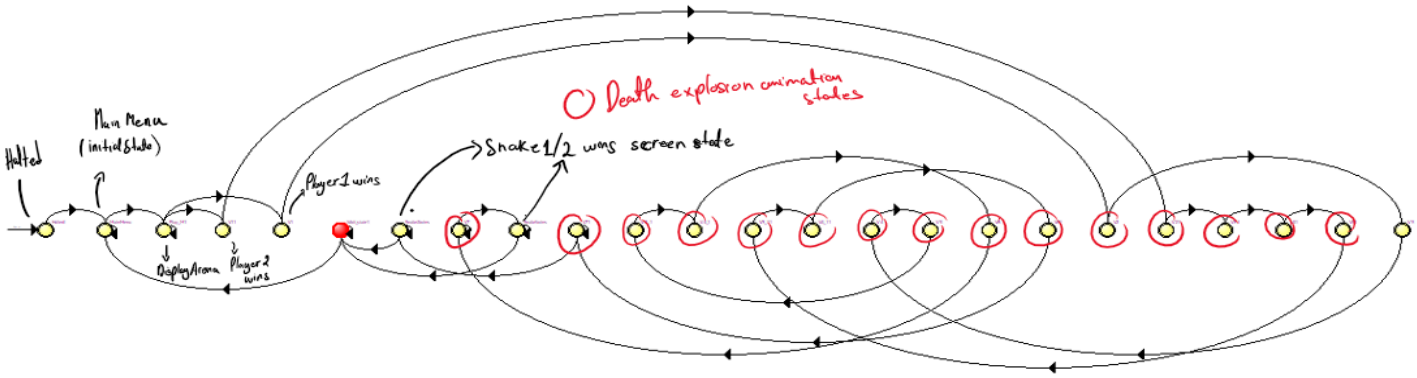


Figure 9: ISDU State Machine

Module: up_head_p1, up_head_p2, right_head_p1, right_head_p2, left_head_p1, left_head_p2, down_head_p1, down_head_p2
Inputs: clock, [9:0] address
Output: [3:0] q
Description: These module take in an address, recognizes which one of the 8 heads it refers to, then sends that signal back to the state machine when a certain keycode is pressed
Purpose: The state machine will recognize which keycode is pressed, then the appropriate snake head will display on the screen along with the direction

Module: venom.sv
Inputs: [15:0] keycode, [7:0] expectedKeycode, [9:0] snakeX, [9:0] snakeY, [1:0] venomCount, [1:0] venomCountState, [1:0] motionFlag, collision, venom_on
Output: venomMovement, [9:0] VenomX, [9:0] VenomY, [9:0] VenomS
Description: Instantiates three instances of venom (3 shots per player) and determines when it hits a player or obstacle
Purpose: We implement a state diagram that uses the ball module from Lab 6.2 to determine when it hits a wall, bound, or player

Module: venomCountMachine.sv
Inputs: Reset, [15:0] keycode, [7:0] expectedKeycode, reload
Output: venomCount

Description: Instantiates three instances of venom (3 shots per player) and decrements the venom count by one when a specific keycode [ENTER] | [SPACE] is pressed for player 1 and player 2 respectively. We included wait states between each venom count transition so that key presses from the keyboard are not double counted causing misfire of venom. Ie. state transitions happen at the falling edges of keypresses. Once all three venom shots are used by the snake (in reload state), the snake can eat an apple that sets the venom state back to Venom1 (meaning it has 3 shots to shoot again).
Purpose: We implement this state diagram to determine the venom count a player has throughout the game, including when a fruit is eaten

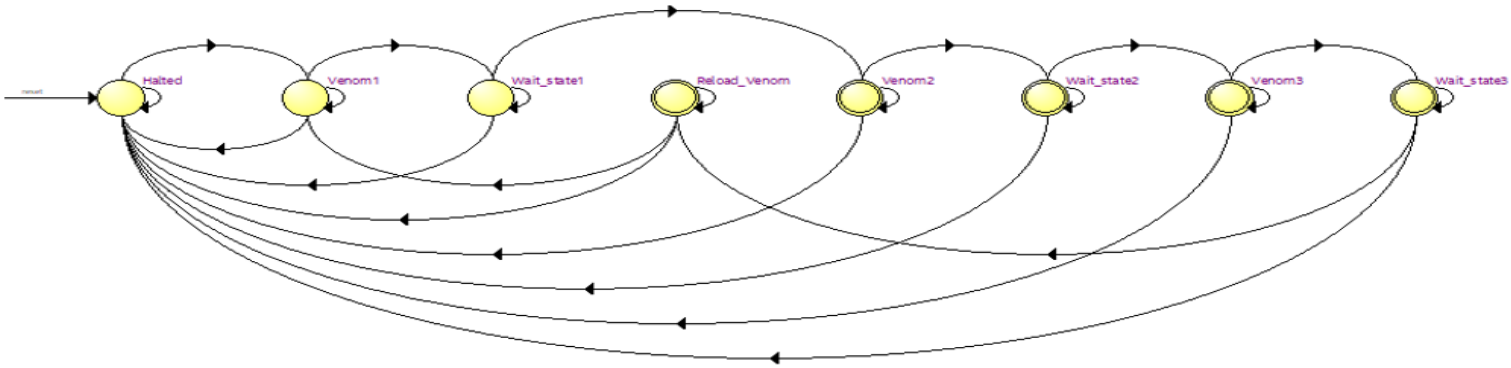


Figure 10: venomCountMachine state machine

Module: venom_stateMachine
Inputs: Reset, [15:0] keycode, [7:0] expectedKeycode, collision, [1:0] venomCount, [1:0] venomCountState, [1:0] motionFlag
Output: [1:0] bulletDir, venomMovement, LED
Description: This module takes in specific keycodes to determine whether a venom is fired, moving, or collided
Purpose: We use this module extensively to understand what is happening to the venom in real time. Due to problems with the bullet changing direction due to user input, we had to store the value of the last keycode

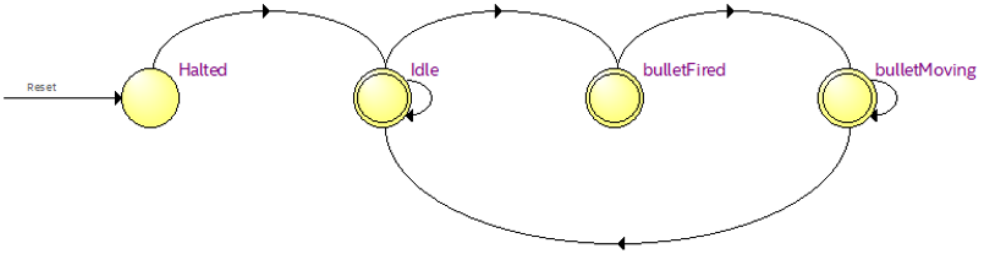


Figure 11: venom_stateMachine diagram

Module: snake.sv/snake2.sv
Inputs: Reset, frame_clk, [15:0] keycode, [8:0] x_velocity, [8:0] y_velocity, [1:0] motionFlag, OB1Flag
Output: [9:0] BallX, [9:0] BallX, [9:0] BallS,
Description: We input the ball location to create the motion flag the snake needs to move. In addition, we added support for collision logic such that the snake will stop when it hits an obstacle.
Purpose: Instantiates snake logistics, such as position, speed, color, and other things we need for the state diagram, such that collision works as intended

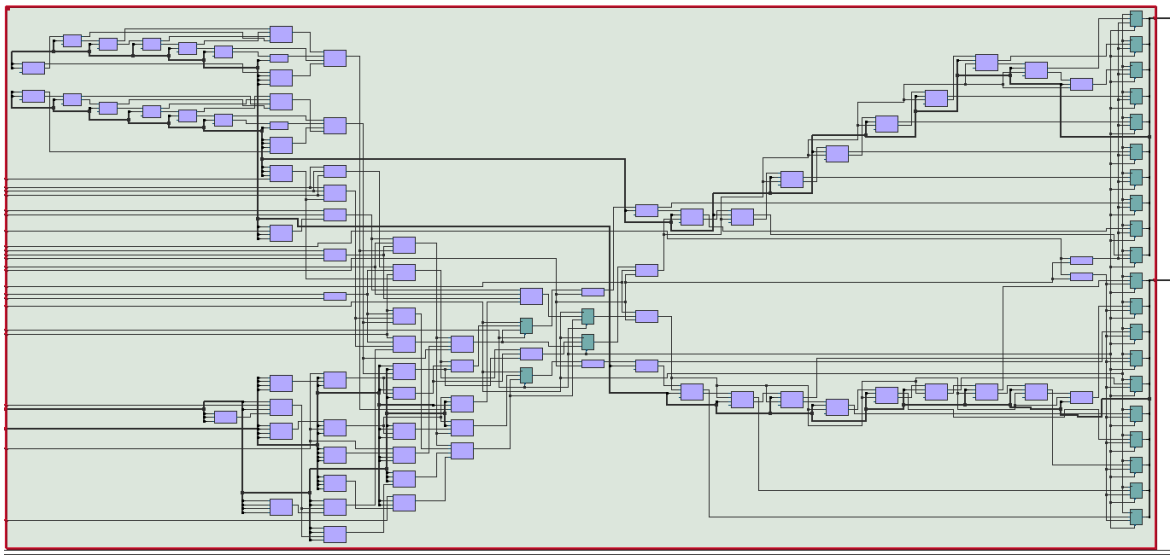


Figure 12: snake.sv block diagram

Module: EasterEgg_rom.sv/Me_Sprite_rom.sv, VenomAnimationSprite_rom
Inputs: clock, [6:0] address
Output: q
Description: Instantiates an ROM we use to generate the easter egg sprite and detection.
Purpose: Synthesizes an EasterEgg.mif file and a sprite of Amaan when a player collects an easter egg.

Module: health_stateMachine
Inputs: Clk, Reset, Collision
Output: [1:0] healthCount, gameEnd
Description: This state machine is instantiated two times for the two snakes to keep track of their lives. The state machine begins at the Halted State upon Reset and sets each snake's healthCount to 2'b11, indicating that the snakes have all three of their lives. The state machine takes a collision signal (snake_on and venom_on at the same time) as input to mark drop in health and

move to the next state which outputs a lower healthCount (ie. 2'b10 for 2 lives, 2'b01 for one life). The last state Health0 is used to mark the end of the game and send a signal to ISDU (our main game state machine) to move to animation and end screen. It can be noted in the state diagram below that we have multiple wait states. This is because we wanted to make sure that collisions between venom and snakes were not being double counted. Essentially, transition between the different health states only happens once the collision flag goes back low.
Purpose: The health state machine is essentially used to keep track of each snake's life count status. In our game, each snake begins with three lives which it loses upon being hit by venom from the opponent snake.

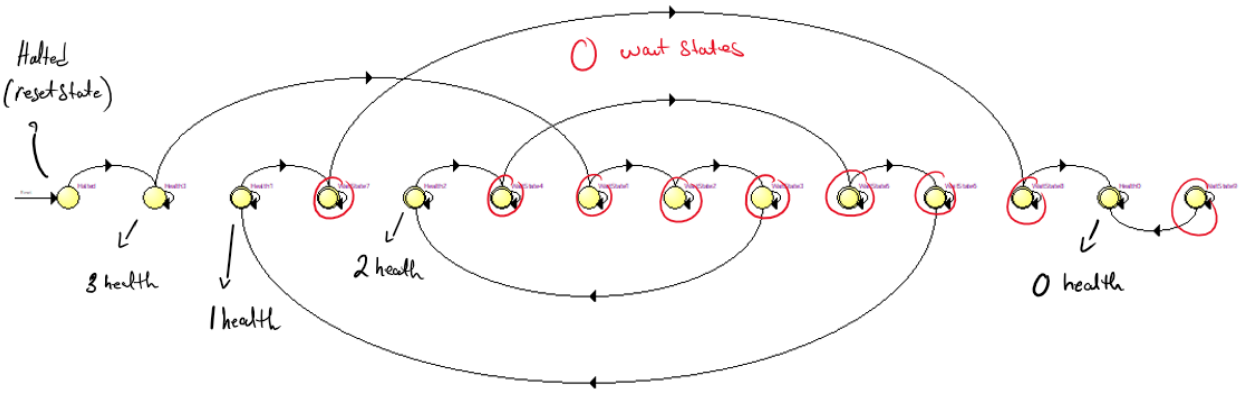


Figure 13: health_stateMachine diagram

Module: move_stateMachineS1
Inputs: Clk, [15:0] keycode
Output: [1:0] motionFlag, Load
Description: This module takes in keycodes from the user and stores it by setting motion flags.
Purpose: These motion flags are crucial for our motion controls since once a key is let go, the state machine remembers which keycode was last pressed and doesn't resort back a default direction

Module: second_counter
Inputs: clk_60Hz
Output: [31:0]
Description: This module counts to 32 seconds
Purpose: We use this module to determine how long until the easter egg shows up

Module: Obstacle_Set1_rom/Obstacle_Set2_rom, EasterEgg_rom.sv/Me_Sprite_rom.sv
Inputs: vga_clk, [9:0] DrawX, [9:0] DrawY, blank
Output: [3:0] red, green, blue
Description: Instantiates color palette that holds the info for the rom file
Purpose: Receives signals from the state machine then returns the appropriate rom file

Module: VGA_RandomCoords
Inputs: Clk, reset, seedIn
Output: [9:0] rnd
Description: This module counts up to 2^32 bits extremely quickly (resetting everytime it hits the upper limit, detects when any keycode is pressed, then sends that bit seed to the state machine for random generation.
Purpose: We use this for random generation of fruit by sending a random seed at a specified time (time of keycode press) to generate a random coordinate at which the fruit will spawn

```
module VGA_Random_Coords (
  input Clk,
  input reset,
  input seedIn,
  output logic [9:0] rnd
);

logic feedback;
assign feedback = random[9] ^ random[6] ^ seedIn;
logic [9:0] random, random_next, random_done;
assign random_next = {random[8:0], feedback};
logic [3:0] count;

always_ff @(posedge Clk or posedge reset) begin
        if (reset) begin
                random <= 10'hF;
                count <= 4'd0;
        end
        else begin
                random <= random_next;
                count <= count + 1;
                if (count == 4'd9) begin
                        count <= 0;
                        random_done <= random;
                end
        end
end
assign rnd = random_done;
endmodule
```

Figure 14: LSFR implementation for random generation

The above code is our implementation of an LFSR. So when we needed random coordinates to be generated for our apples on screen we decided to use an LFSR (Linear Feedback Shift Register). An LFSR is simply a shift register that has some of its bits (known as taps) XOR'd with themselves to create a feedback term. One major factor we had to account for when implementing an LFSR was its width and repeatability. After much trial and research we discovered that an N-bit LFSR will be able to generate $2^N - 1$ random bits before it starts

repeating. In an LFSR the MSB will always be the feedback point. The main thing we had to take care of while coding the LFSR was to know which bits are the taps (to be selected for XOR). This was confusing as the taps are different for different size registers. Thankfully Xilinx had calculated optimum taps and made it available to us in its online documentation (Xilinx XAPP 210 Linear Feedback Shift Registers in Virtex Devices, v 1.0 (8/99) (hu-berlin.de)) However, the level of randomness was not quite enough for us and so we decided to use a seed input that would be used as a tap to randomize our inputs even more. In our case the seedIn was a bit value of a counter value that is selected upon user key presses. With this we were able to achieve a level of randomness that was acceptable.

Collision Logic

```
// SETTING HIT OBSTACLE FLAG FOR SNAKE 1
 logic [9:0] snake1X, snake1Y;
 assign snake1X = DrawX - snakeX_pos;
 assign snake1Y = DrawY - snakeY_pos;

 always_ff @(posedge vga_clk) begin
       if (DrawX == 0 && DrawY == 0) OB1Flag <= 1'b0;
       if (!(paletteOb1_red == 4'hF && paletteOb1_green == 4'h0 &&       paletteOb1_blue ==
4'hF)) begin

              case (motionFlagOut)
                     2'b00 : begin         // W
                                   if (snake1X < 24 && snake1Y == '1) begin
                                                 OB1Flag <= 1'b1;
                                   end
                              end
                     2'b01 :begin          // A
                                   if (snake1X == '1 && snake1Y < 24) begin
                                                 OB1Flag <= 1'b1;
                                   end
                              end
                     2'b10 :begin          // S
                                   if (snake1X < 24  && snake1Y == 24) begin
                                                 OB1Flag <= 1'b1;
                                   end
                              end
                     2'b11 :begin          // D
                                   if (snake1X == 24 && snake1Y < 24) begin
                                                 OB1Flag <= 1'b1;
                                   end
                              end
       end
 end
```

Figure 15: LSFR implementation for random generation

As mentioned previously, our background and obstacle course are two different sprites. This was done not only for game scalability but also to make collision detection easier to implement. The above code snippet shows our logic that raises a flag (OB1Flag) once collision is detected. The way the logic works is by first checking whether the electron gun of the monitor is drawing the pink region of the obstacle course. Not drawing pink would indicate that it's drawing some obstacle on the screen. We then proceed to check if the snake is within some distance from the obstacle (square box 24x24 pixels within which the snake is drawn) depending on its motion. If it is, then the collision flag is raised and the signal passed into snake.sv to stop the snake's motion. We also make sure to reset the collision flag at every frame so that the snake is able to resume motion in another direction. The same logic is then repeated for the second snake.

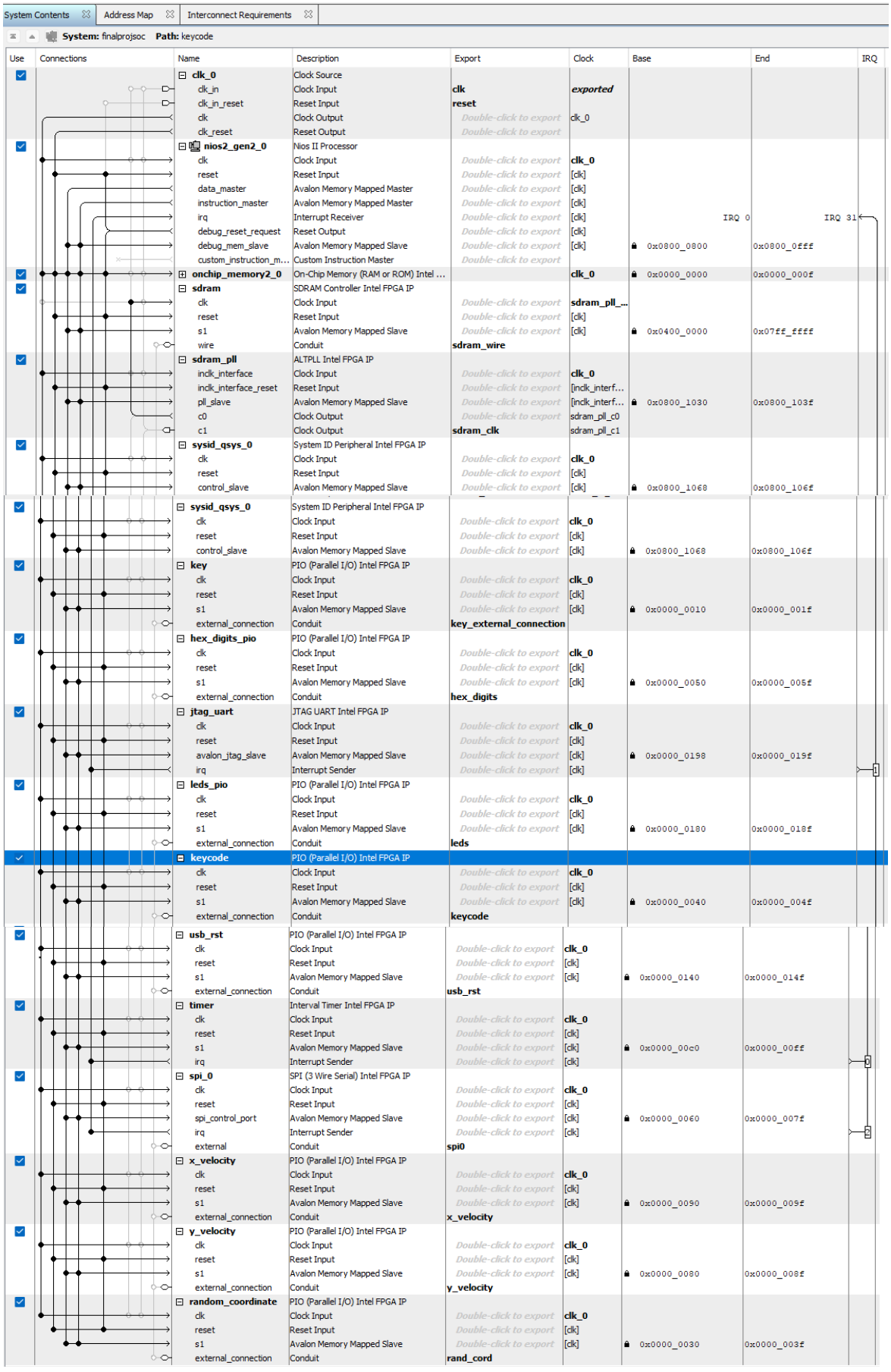
Glitches and Issues Faced

One of the major problems we encountered was in the form of timing and clock variation issues. Because our project involved the use of multiple clocks (50MHz, 25MHz, and 60Hz) we had to make sure that there were close to no difference in signal propagation times between different parts of the digital circuit. Towards the final phase of our project we encounter issues such as race conditions, setup and hold time violations, and metastability. One specific example is that our collisions were being double counted due to the fact that the collision flag remained high for more than one clock cycle. To resolve this issue, we had to make use of a clock divider to create a custom clock that worked in some state machines while also adding additional wait states in other state machines to wait for the right time to switch states.

One other issue we faced was handling collisions between walls and snakes. We had originally implemented our drawing logic in such a way that all objects were drawn relative to the center. This meant that we somehow had to find a way to make negative number comparisons when using the collision logic mentioned above. Because it is a little extra complicated to handle negative numbers in hardware, we decided to rewrite our draw logic so that we were drawing from the top left corner instead of the center. This made collision detection much easier to implement and understand.

Last but not least, we had initially just written every line of code within the color mapper itself and towards the mid phase of our project it seemed that things were getting out of order and we had to optimize our file structure.

Platform Designer



Overall our hardware implementation makes use of the SOC that was generated in lab 6 which include the following components.

- clk_0: The main clock that the FPGA, MAX3421E, SDRAM needs to run on to run synchronously.
- nios2_gen2_0: This is the 32-bit CPU that is controlled by our software (using C programming language). Used to control peripherals and handle transmission of data between them.
- sysid_qsys_0: Checks and returns whether the system ID is correct so it correctly executes software onto the hardware.
- [peripheral]_pio: The base address of these components (ex. Keys, leds, switches, and Hex) are set in Platform Designer so that we can access them to use in our software.
- spi_0: The SPI (System Peripheral Interface) lets us interact with USB Peripherals like the Keyboard, and mouse.
- jtag_uart: This allows data communication movement between the computer and FPGA that allows for the transferring of text which avoids slowing down of the CPU for tasks that don't need high processing power. This includes functions such as printf for debugging purposes.

One change we had to account for however is the fact that we were taking input from two different users on one keyboard peripheral. This meant that we had to extend our keycode PIO to 16 bits from 8 bits. This change is also accounted for in the C driver controller for the MAX3421E chip. The change made is documented below.

```
void setKeycode(WORD keycode1, WORD keycode2)
{
       IOWR_ALTERA_AVALON_PIO_DATA(0x40, (keycode1 << 8) | keycode2);
      // Shifting keycode1 into upper half and adding keycode2 to lower half
}
```

So the setKeycode function was used to write each keycode input value (upper half begin keycode1 and lower half keycode2) to the memory address assigned to keycode in platform designer.

Top Level RTL View:

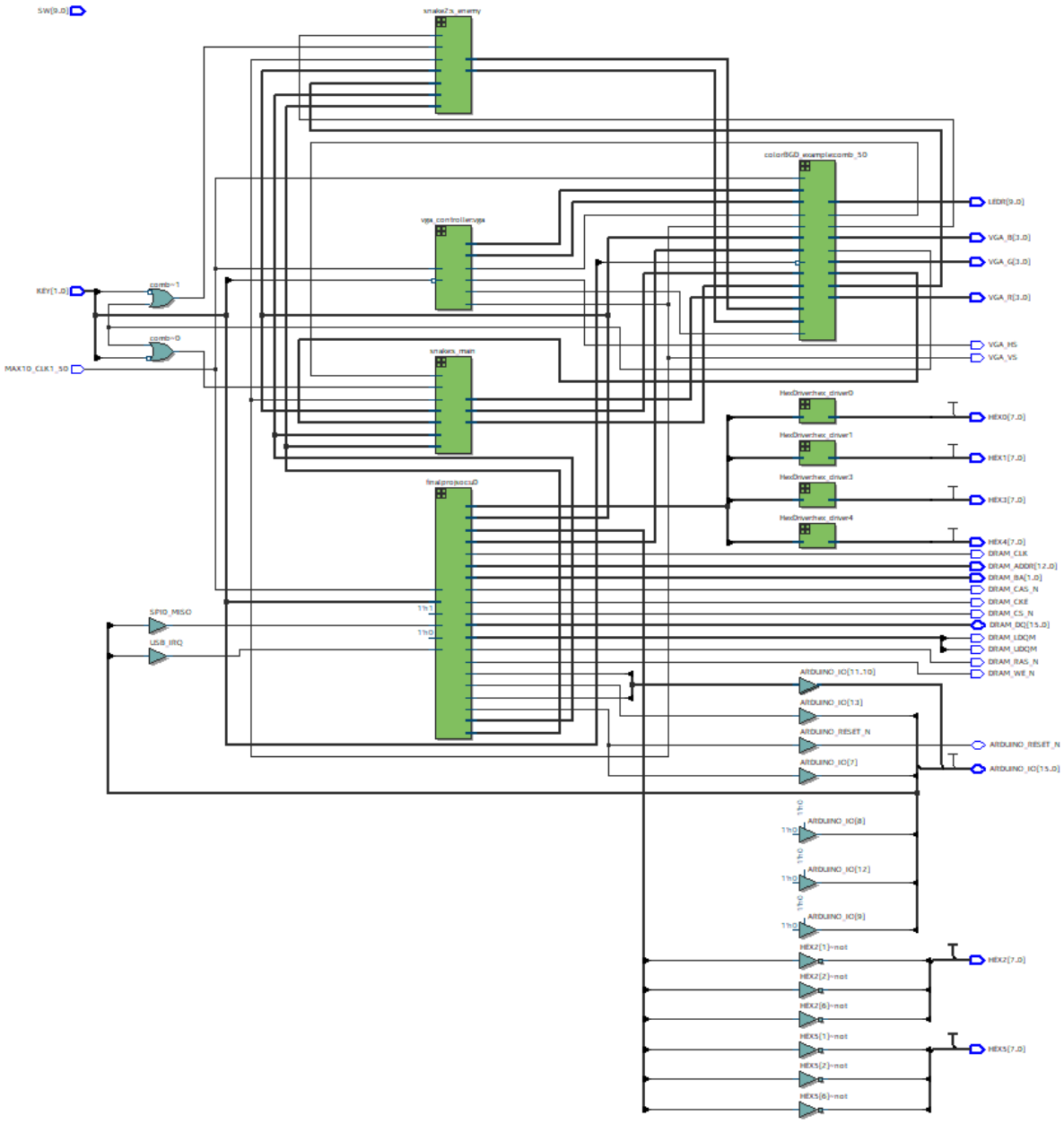


Figure 16: Top Level Viewer

Design Resources and Statistics

LUT	6540
DSP	12
Memory (BRAM)	786,784 bits
Flip-Flop	3256
Frequency	106.22 MHz
Static Power	96.18 mW
Dynamic Power	0.72 mW
Total Power	106.22 mW

Figure 17: Final Project Resource Usage and Summary Statistics

Conclusion

This final project we created was only possible because of all the labs prior to this. Without those fundamental building blocks, our success in this project may not have been achievable. From Lab 5, we used the idea of MUXes and other components in SystemVerilog to aid with our state machine and various other logic. We used Lab 6.2's code excessively with our motion controls and establishment of a real-time communication between hardware and software. Finally, we used Lab 7's use of fontrom for our additional sprite, snake heart count/projectile count, color palette, and sprite handling logic with on-chip memory.

The most difficult part of this project was figuring out the rest, such as finding various tools we can use to get our game logic working. Without all the CAs, other struggling students, and the helper tool, I'm sure we would have taken a lot longer to get to our baseline difficulty.

However, at the same time, I wish we had just a bit more time. After getting our base game working, we felt as if our project was very scalable. In addition to the simple features we added, we could have made animations as elaborated earlier, we could have added different game modes and difficulties that required only the slightest of changes in game logic, added mouse control support, audio (we had the working file but wasn't able to show it), and so much more. This is completely within our grasp, especially since we only had used 45% of memory for our demo.
We learned so much from this project, from photoshop, memory management, and hardware random generation, to state machines, sprite handling, and helper tool logic.