FPGA Snake Game

Eric Wyman, Peyton Schmid
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: ewyman@oakland.edu, peytonschmid@oakland.edu

***Abstract*— The purpose of this project is to create a snake game using a Nexys A7 FPGA board. The game utilizes a keyboard input for directional control and a VGA output to display the game on a computer monitor.**

I. Introduction

This application was implemented with an asynchronous keyboard input that was converted into simpler logic for user-inputted control. This input was used to calculate the next state of the game, which includes initialization, snake movement, snake lengthening, the production of food tiles, and finally the "game over" state. This logic ensures that the displayed snake movement complies with the rules of the classic computer game, "Snake." From here, the game state logic is converted to a VGA output display using 12-bit RGB color.

The primary motivations for this project are to better understand the interface between a computer and its keyboard or display and to understand the process of implementing a basic game using a hardware description language like VHDL. To finish this project, knowledge on finite state machines, counters, and registers from class were used. It was also necessary to learn more about different data types in VHDL and the innerworkings of PS/2 and VGA communications.

II. Methodology

A. VGA Display

The snake game requires a display to show the user what is happening. A VGA display was chosen because VGA control is relatively simple, and all of the required hardware is built in to the Nexys A7 board.

To stay within the time constraints of the project, Prof. Llamocca's vga_ctrl code from the class website was used [1]. After looking through this code, it was determined that the simple color configurations were better suited for this project than the memory configuration. The focus of the project was not on graphical detail so using memory to store pixel data would add unnecessary complexity. When configured to the simple 12-bit color mode, vga_ctrl generates several signals that are important for the rest of the game. Vga_ctrl uses a 12-bit vector called SW to set the color value for pixels. In the original code this is mapped to on-board switches so a user can enter a 12-bit color with switches and display that color across the entire VGA monitor [1]. For this project, control over individual pixels was necessary to display the game. To accomplish this, three different signals were used: hcount, vcount, and vga_clk. Hcount and vcount are basic 10-bit counters that keep track of the horizontal and vertical pixel count. Together they form pixel coordinates ranging from (0,0) at the top left corner of the screen to (639, 479) at the bottom right. Vga_clk is a clock used for pixel color timing. On every pulse of the clock, the color of the pixel at the location (hcount, vcount) is written from the SW vector. To draw the different colors on the screen, the game needed to write appropriate color values to SW on this vga_clk pulse.

The gameDisplay block is responsible for this task. This block uses the locations of each piece of the snake, the food, and the walls to determine the correct color for a pixel and writes that value to SW on the vga_clk.

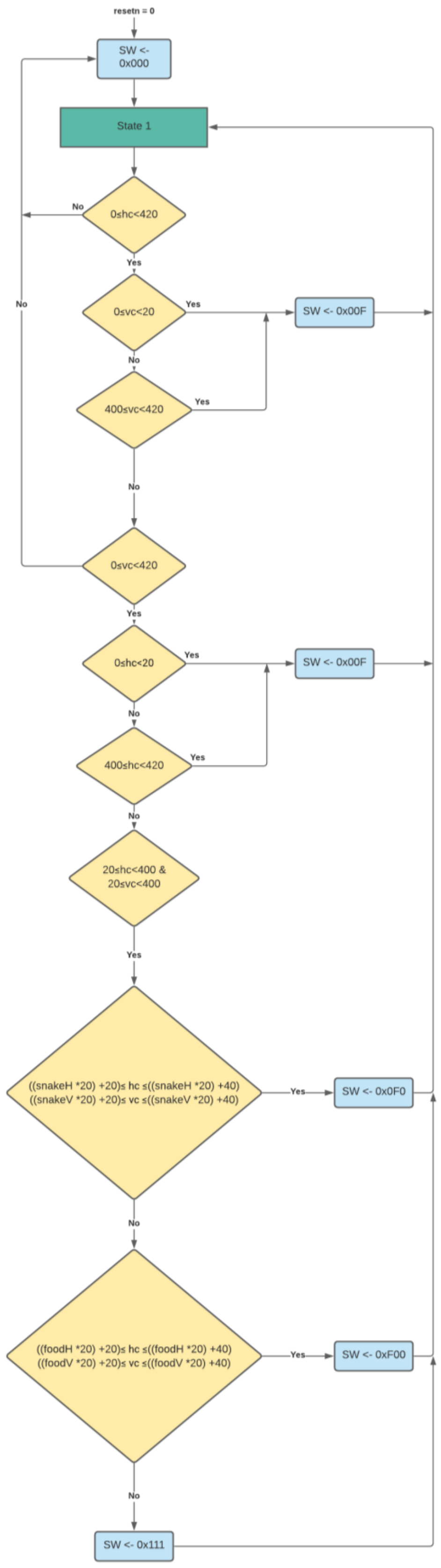
B. gameDisplay

To keep the project simple, a square play area was used. The play area (not including walls) spans from pixel (20, 20) to pixel (399, 399). This play area was divided into a 20x20 grid of squares that are each 20x20 pixels. This allows for the game logic to use a smaller "snake" coordinate system to map the positions of the snake and food that can later be converted to their respective pixel ranges. For example, a snake piece at position (1,1) on the 20x20 grid would have a snakeH value of 1 and a snakeV value of 1. When these values are sent to the gameDisplay block, they are converted to min and max horizontal and vertical pixel count values using the following equations.

$$hc_{min} = snakeH * 20 + 20$$
$$hc_{max} = snakeH * 20 + 40$$

When hcount and vcount are within the range of values for the snake, the color green (0x0F0) is written to the SW vector on the vga_clk tick. Every piece of the snake and the food is converted and displayed in this way. Figure 1 below displays the ASM chart for this logic.

Figure 1: ASM Logic gameDisplay



C. *PS/2 Keyboard Interface*

A USB keyboard was integrated as the input to allow for user control of the game. Digilent FPGA boards are compliant with PS/2 protocols and can seamlessly convert input data from ASCII characters as input functions to a program [2]. A USB keyboard can also use this same PS/2 logic with some coding conversions and a debounce function, as the scan codes are sent from the keyboard every 100 milliseconds. The code used in order to convert the logic of the keyboard to a readable input by the FPGA was implemented through Professor Daniel Llamocca [1]. By applying this code, the keyboard input was further simplified due to the five keys that are used to control the Snake application: the four arrow keys and the space bar. The arrow keys were parsed into a 2-bit binary vector, while the spacebar was set to a one-bit binary value that is high when pressed, and low when unpressed. Table 1 below displays the conversion values for the arrow keys.

Table 1: Keyboard Movement Input Conversions

Up Arrow Key	“00”
Right Arrow Key	“01”
Down Arrow Key	“10”
Left Arrow Key	“11”

From here, the PS/2 input is implemented into the logic of the game. Figure 2 displays the ASM diagrams that explain the operation of the PS/2 interface, while Figure 3 displays the top diagram of the PS/2 converter component.

Figure 2: PS/2 ASM Diagram

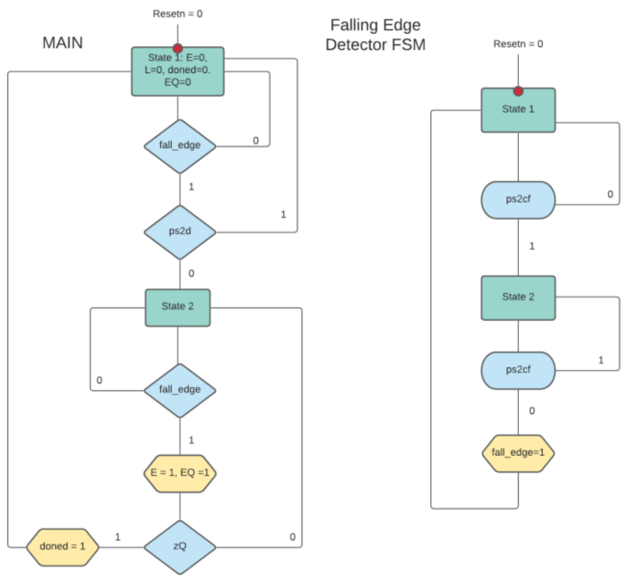
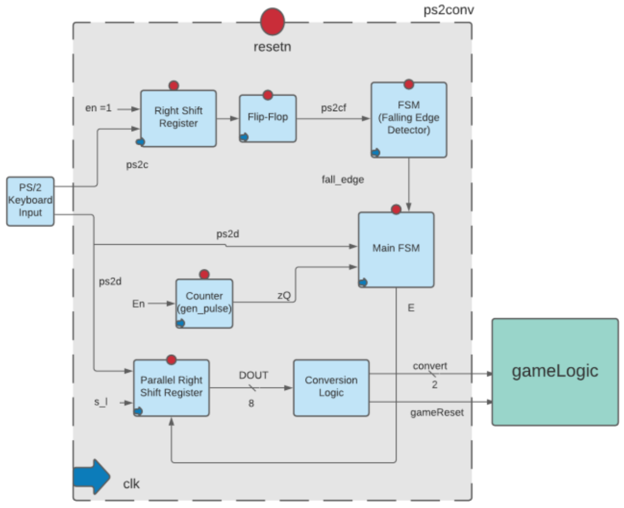


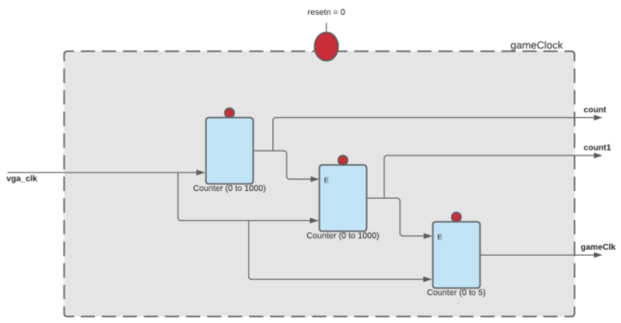
Figure 3: PS/2 Top Diagram



D. Game Clock

The gameClock block is used as a basic clock divider to bring the 25MHz, vga_clk down to a much lower speed for the game to update at. After some experimentation, it was found that around 5Hz or 5 movements of the snake per second was an appropriate speed to run the game at. The counters used to divide the clock are also used for the pseudo random food generation. Figure 4 below shows the operation of the game clock.

Figure 4: gameClock Diagram



E. Game Logic

The game logic is comprised of 6 different game states, each handling different functions of the game. This block uses two special data types. The first is called snake, this is a 15 slot array of integers that holds one half of the location of the snake pieces in the 20x20 coordinate system. The size of this array sets the limit on the length of the snake. The other data type is state which can have one of the following values: start, getDir, checkTile, move, food, and gameOver. The state data type is used to set up the main gameLogic state machine.

The game starts in the "start" state. The only purpose of this state is to hold the current positions of the snake until the next tick of the gameClock. Without this state, the game would run at the speed of the vga_clk, so the snake would be moving at 25Mhz which is far too fast to be a playable game.

After the gameClock tick, the logic moves to the "getDir" state which writes the current keyboard input to a direction variable to be used in the next state.

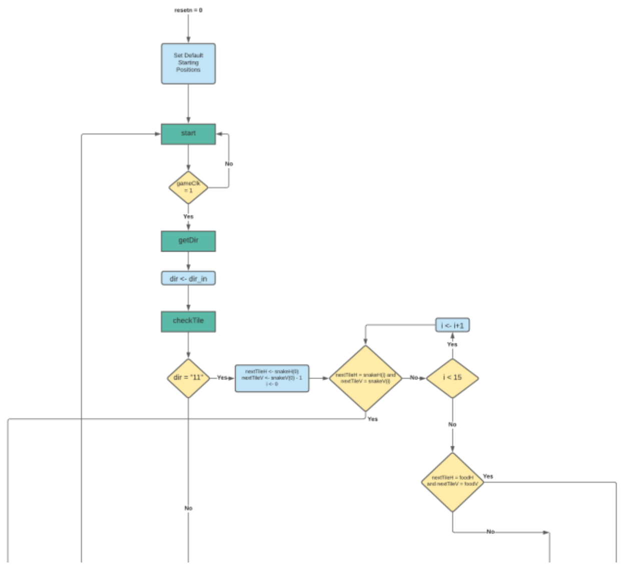
On the next tick of the vga_clk, the game moves to the "checkTile" state. In this state, the head of the snake is used with the direction input from the last state to determine the position of the tile the user is trying to move to. This position is stored in the nextTile variable. This nextTile position is then compared to each tile of the snake, the walls, and the food to determine whether the game will move to the "move", "food", or "gameOver" state.

If the nextTile is the same as the current food tile, the game moves to the "food" state where the snake length variable is increased, and a new food position is obtained from the pseudo random food generator. In this state, the logic checks to make sure the new food position is a valid one (not the same as the current position or inside the snake) and moves to the "move" state when the new food position is valid.

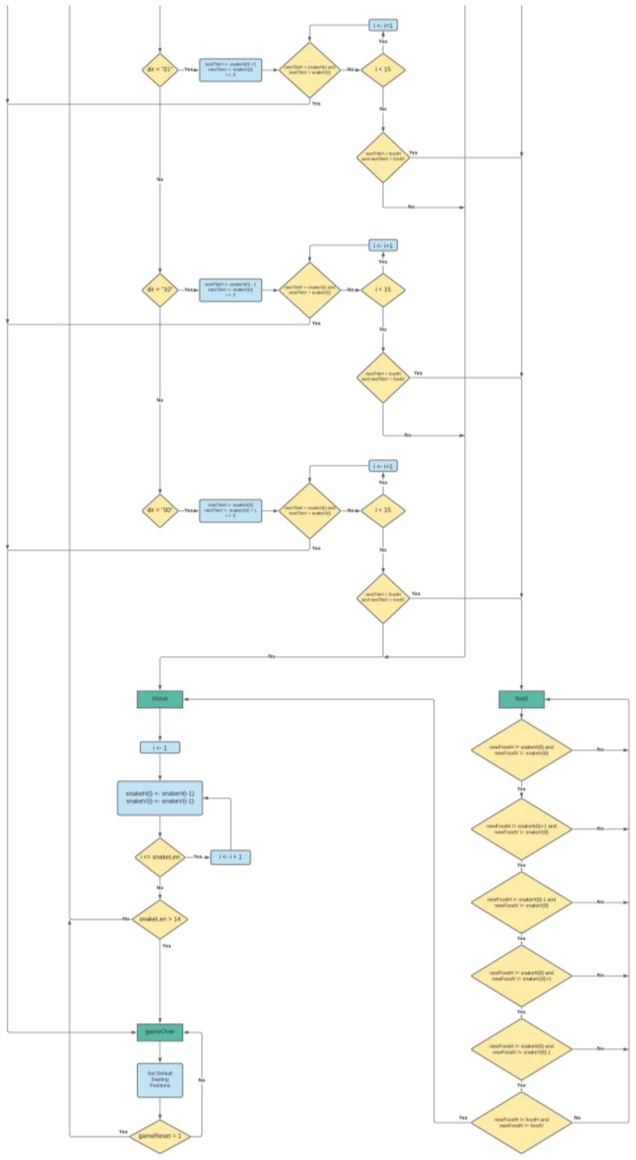
In the "move" state, the positions of each piece of the snake are shifted and the head is moved to the targeted next tile. After moving the snake positions, the game loops back to the "start" state and waits for the next tick of the gameClock.

If a wall or snake is hit or the snake has gotten too long, the game enters the "gameOver" state, where everything is reset to the starting positions and the game waits for the spacebar to be pressed to restart the game. Once the spacebar is pressed, the game loops to the "start" state. Figure 5 below illustrates the full ASM chart for the logic of the game.

Figure 5: gameLogic ASM Chart



resetn = 0
Set Default Starting Positions
start
No
gameClk = 1
Yes
getDir
dir <- dir_in
checkTile
i <- i+1
Yes
dir = "11"
Yes
nextTileH <- snakeH(0)
nextTileV <- snakeV(0) - 1
i <- 0
nextTileH = snakeH(i) and nextTileV = snakeV(i)
No
i < 15
Yes
No
nextTileH = foodH and nextTileV = foodV
Yes
No
No



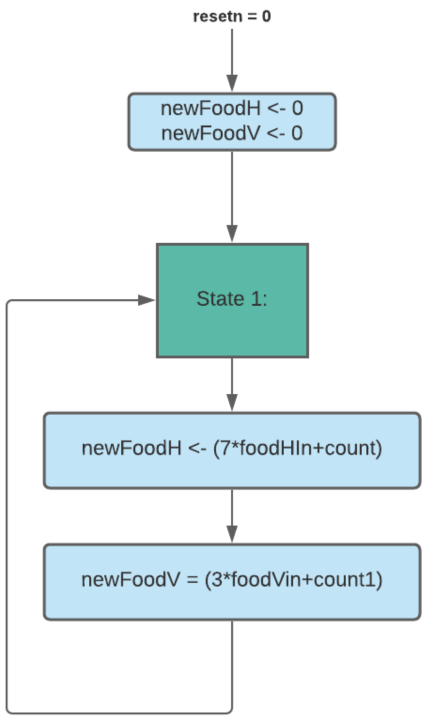
F. *Pseudo Random Food Generator*

The genFood block generates a new food position on every tick of the vga_clk. This new position is only ever used when the game logic is in its "food" state, meaning the current food has just been eaten and a new piece needs to be generated. The equations used to generate new locations are as follows:

$$newFoodH = (7*foodHIn+count) \mod 20$$
$$newFoodV = (3*foodVIn+count1) \mod 20$$

Where foodHIn and foodVIn are the current food coordinates, 7 and 3 are arbitrarily chosen multipliers, and count and count1 are counters from the clock divider. The modulo operator is used at the end to convert the large random values to a scale between 0 and 19 that fits the 20x20 tile coordinate system. Figure 6 below demonstrates the ASM logic behind the pseudo-random food generator.

Figure 6: Pseudo-Random Food Generator ASM Chart



III. EXPERIMENTAL SETUP

The function of the application was verified using progressive trial and error. First, the VGA display code was tested using a Nexys A-50T FPGA board. Switches were used to ensure that the screen had a full functioning range of color that corresponded with the correct designated 12-bit vector. From here, the game logic was implemented without the PS/2 keyboard. The provided switches on the FPGA board were used to control the direction input. The code was troubleshooted and tested by hand in order to ensure that the game complied with the operation of the classic snake game. From here, the keyboard was implemented. The code was edited until the keyboard input seamlessly worked with the rest of the project. Once the structure of the project was verified, slight edits were made to the playing field to provide a more aesthetic and familiar set up to this rendition of the classic game. For example, the snake was changed to a green color and the generated food was set to red, while the game field was set to gray with a blue border.

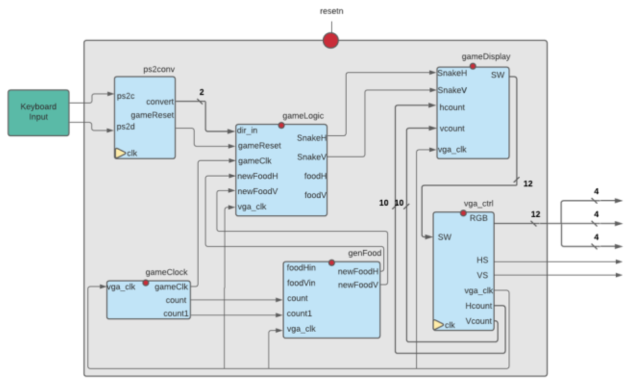
IV. RESULTS

The program was implemented seamlessly, although a bit rough around the edges in terms of visual design. The controls of the game allowed for proper function, and the display provided a visual rendition of the input for game operation. When the game was "lost" due to a collision with either the border or the snake itself, the game was able to be reset and played again by pressing the spacebar key. The link below redirects to a video of the function of the project:

https://www.youtube.com/watch?v=05jZkSMZmzs

Figure 7 below shows the top-level diagram of the project.

Figure 7: Top Level Diagram of Snake Game



Conclusions

This project allowed the group to delve deeper into the programming of FPGAs using VHDL through the implementation of the PS/2 keyboard as well as the VGA display. Components such as shift registers were utilized within the provided code documents. As these topics were further explored, the implementation of the logic units that were discussed within the class became clearer. The project also allowed for skills of troubleshooting and implementation to become stronger, as this project was implemented out of creativity and not just from simply following a laboratory rubric. Although the application ran quite seamlessly, more could be done to make the application more visually aesthetic. For example, a playing grid with a greater density of pixels could be use for higher resolution graphics in both the snake and the generated food. More detail could have also gone into the playing ground to make it more visually pleasing as well. Different difficulty levels could also have been added by increasing the snake speed through the game clock. A scoreboard would also be a pleasing addition so that the user could attempt to beat their high score.

References

[1] Llamocca, Daniel. "RECRlab - VHDL Coding for FPGAs." *Reconfigurable Computing Research Laboratory (RECRLab), Electrical and Computer Engineering Department, Oakland University*.

[2] Brown, Arthur. "Nexys A7 Reference Manual." *Nexys A7 Reference Manual [Digilent Documentation]*, Digilent, reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual.