Neil Marks

ECE350 Final Project Report

GitHub Repository: https://github.com/neilmarks13/ECE350FinalProject.git

The game Snake is one of the classics of video games. It is a simple game where the player navigates the snake in order to eat apples while avoiding hitting the rest of the snake or the borders. With each apple that is eaten, the snake grows in length making it more difficult to navigate. The goal of the game is to eat as many apples as possible without hitting anything.

For my final project, I used my processor, with multiple other tools integrated, to run a custom MIPS program to play the game Snake. In this version of the game, the snake moves after a fixed number of clock cycles as specified by the MIPS program (about 33 cycles). The movement of the snake is controlled by the A/D/W/S keys to move left/right/up/down; if no key is pressed or if the backwards direction is pressed (i.e. the snake is moving left and the right key is pressed), then the snake continues moving in the same direction as it is. The snake head is a 2x2 block of pixels moving around a 16x16 grid. Each time the snake head reaches an apple location, a new apple is then generated, the score is increased by one, and the snake grows in length by four 2x2 blocks of pixels. Just like in other versions of the game, if the snake head hits the rest of the snake or the border, then the game is over and is restarted with the score being reset to zero.

In order to display the game, a video LCD display had to be added to the processor (shown in figure 1 in the appendix). This video display takes as inputs from left to right a reset bit, a clock bit, a write enable bit, a 7 bit X coordinate, a 7 bit Y coordinate, and a 16 bit color code. This video display was positioned in the write stage of the pipeline. A “vid $rd, $rs, N” instruction was added to the instruction set. This instruction writes a pixel of the color specified by the lower 16 bits of N to the location specified by “$rd + $rs” where the lower 7 bits are the X location and bits 7-13 are the Y location.

To communicate with the player on the controls for the game and to display the score, a character LCD display was added as well (shown in figure 2 in the appendix). From left to right, this display takes inputs of a reset bit, a clock bit, an enable bit, a 5-bit location, and an 8 bit character ascii value. A custom instruction of “write $rd, $rs, N” writes the letter encoded by the lower 8 bits of “$rs + N” to the location specified by bits 8-12 of this sum.

To input the keyboard key pressed, the keyboard queue was implemented into my processor (shown in figure 3 in appendix). This keyboard queue takes as inputs from left to right a reset bit, a clock bit, and an enable input, and it outputs the ascii value of the leftmost character in the queue. By establishing the enable bit to always be zero and the reset bit to be high when the keyboard input instruction was called, the characters are never popped off from the left of the queue, and the ascii value output will always be for the first key pressed between moves. The ascii value output was then input into a sub-circuit that had a four-bit output where each bit corresponded to either an A, S, D, or W being pressed for the four directions. These four bits were then used to control a series of multiplexers that would choose the correct direction of travel for the snake. The options were either +2 to go right, -2 to go left, +256 to go down (+2 shifted left 7 bits for the Y direction), or -256 to go up. The direction of movement was +/-2 because the head of the snake was 2x2 pixels, so one movement corresponded to moving 2 pixels. This direction was then stored in a latch between the decode and execute stage and was only updated when one of the four direction keys was pressed. A custom instruction of "key $rd" was established to write the value being stored in this direction latch into $rd.

Finally, a random number generated had to be added to determine the apple location in the game. The Logisim random number generator was used (shown in figure 4 in appendix). At the start of each game, the snake head is initialized at the center location and only the top left corner is tracked. Since the location of this corner is odd in both X and Y coordinates and the snake always moves 2 pixels at a time, the random apple location needs to always be odd hence the fixing of bits 0 and 7 to be ones. An instruction "rand $rd" was implemented to write the randomly generated odd number to $rd.

A MIPS program with these custom instructions was also written for the processor to run. An outline of that code can be seen in a flow chart in figure 5 in the appendix. The code simply follows the game where after initializing it, takes in the move from the keyboard, checks the next location, and either simply moves the snake, eats the apple, or ends the game based on whether the snake hits nothing, an apple, or itself or the border.

The location checking is actually done in two ways. The apple location is kept in a register and is simply compared to the head location after each move with a "bne" instruction. The RAM is used to store items at every other location in the game. With each move, if there is not an apple at the new location, the 14-bit location (7 Y bits followed by 7 X bits) is loaded from memory. If the value is nonzero, then the snake has hit something, and the game is ended. Whenever the snake moves, the direction of movement is stored in RAM at that location. The location of the back of the snake is kept in a register and is used to remove the back segment of the snake when it moves. With the direction of movement being stored at each location, the direction can be loaded for the back location such that the register can be easily updated to the new back of the snake whenever the snake moves. This allows for constant timing in the game no matter how long the snake gets since only the front and back locations are ever updated.

The other crucial part of the code is score updating. A conversion from binary to decimal is done in software where one register is established to hold the ones place of the score and another to hold the tens place. This allows for printing of the score to the character LCD display with the ascii values of the numbers being stored in the registers.

With this program and hardware modifications and after hours of testing through playing the game, it was determined that the game plays exactly as designed. Dealing with the clock in Logisim was challenging in that it never reached a high enough frequency to allow for smooth movement. In order to combat this, the game was scaled down to the version described above that only uses a corner of the video display and the code was optimized to minimize jumps and wasted clock cycles.

Another challenge faced was using the premade keyboard queue. It is not an ideal keyboard input as the keys pressed will build up in the queue especially when a key is held. This is why the queue is wired the way it is with the key instruction clearing the queue each time a direction is determined so that the next key press to move the snake is not missed.

Also, restarting the game after the snake hits something proved to be more difficult than expected. By using the direction latch to store the previous direction traveled, the snake would start moving as soon as the game restart in this direction originally. In order to combat this, a reset of this latch was established by making back to back key instruction reset this latch to zero. This way, two key instructions can be put in the code when restarting the game to reset this latch. This same reset was also used to reset the character display to be able to print new messages quickly such as switching between “Use A S D W to move Score:__” and “Game Over Score:__”.

In all, this project, although slow, is definitely a success. The game plays correctly to the specifications established earlier. With more time, it would have been fun to try to convert this game to a simpler finite state machine that would hopefully run faster on the Logisim clock. Also, making the graphics better would have been a goal as well by making the snake look more like a snake and using the entire video display. The final display of the game can be seen in figure 6 in the appendix.

**Appendix:**

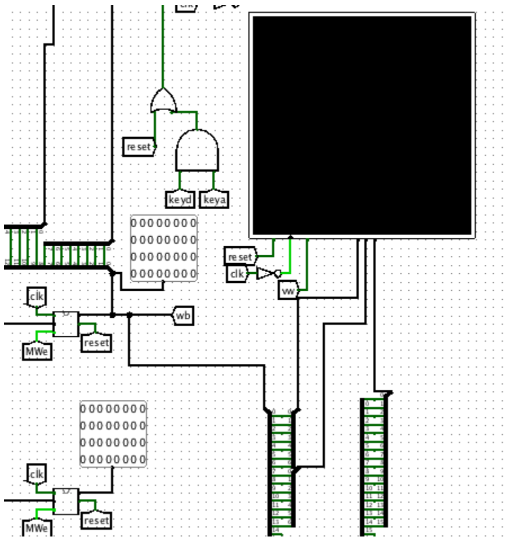

*Figure 1: LCD Video Display*

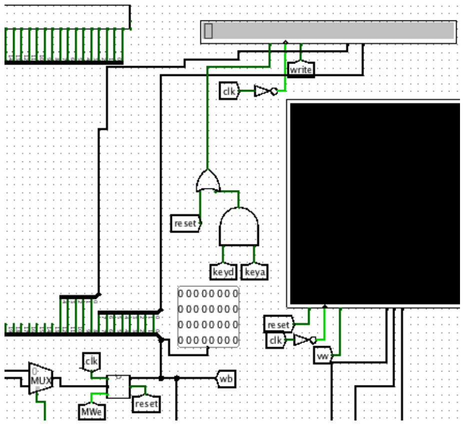

*Figure 2: LCD Character Display*

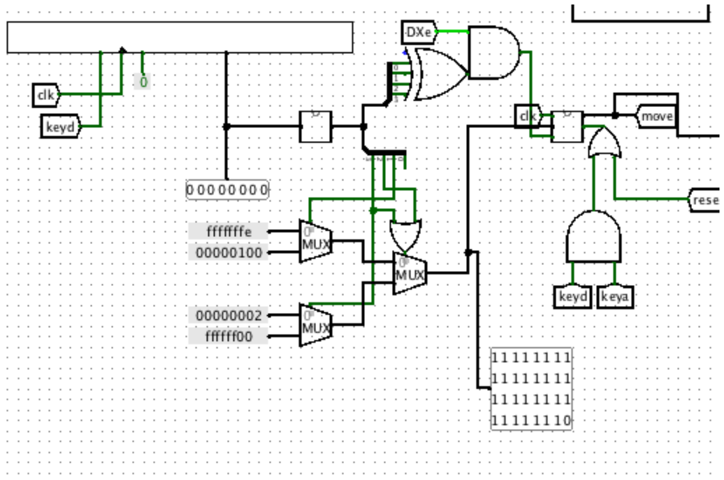

*Figure 3: Keyboard Input and Direction Decoding*

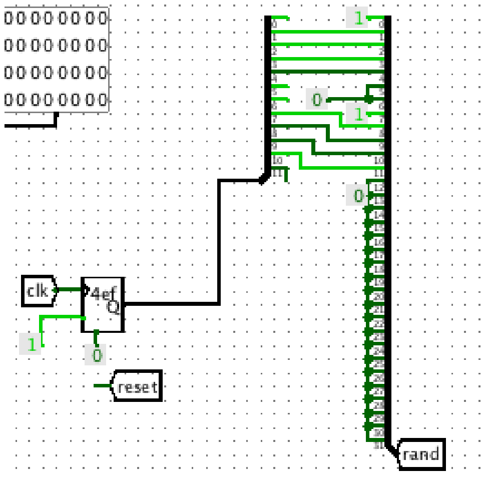

*Figure 4: Random Number Generator*

.

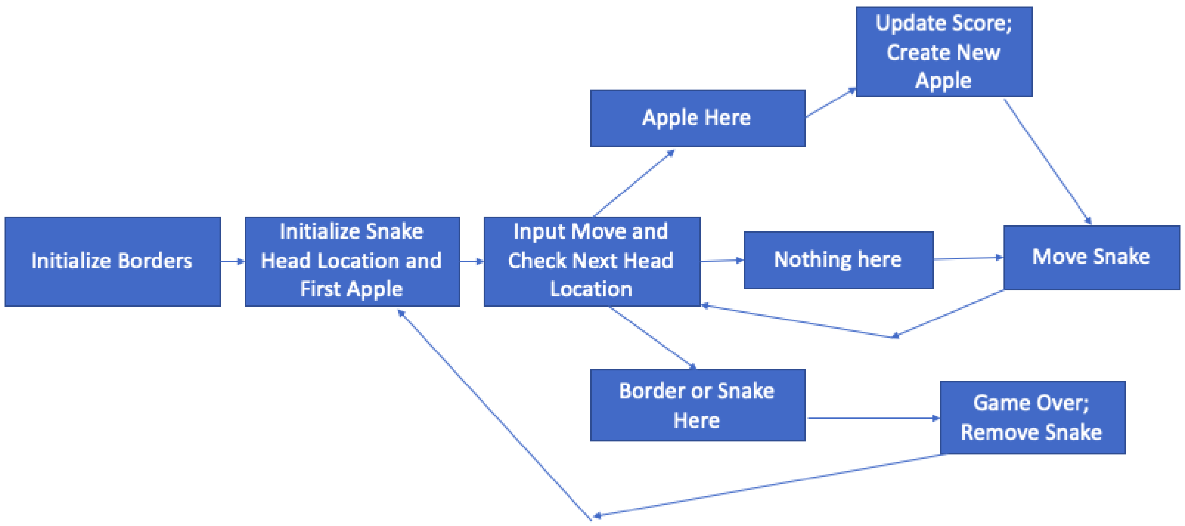
Initialize Borders
Initialize Snake Head Location and First Apple
Input Move and Check Next Head Location
Apple Here
Update Score; Create New Apple
Nothing here
Move Snake
Border or Snake Here
Game Over; Remove Snake

*Figure 5: MIPS Code Flow Chart*

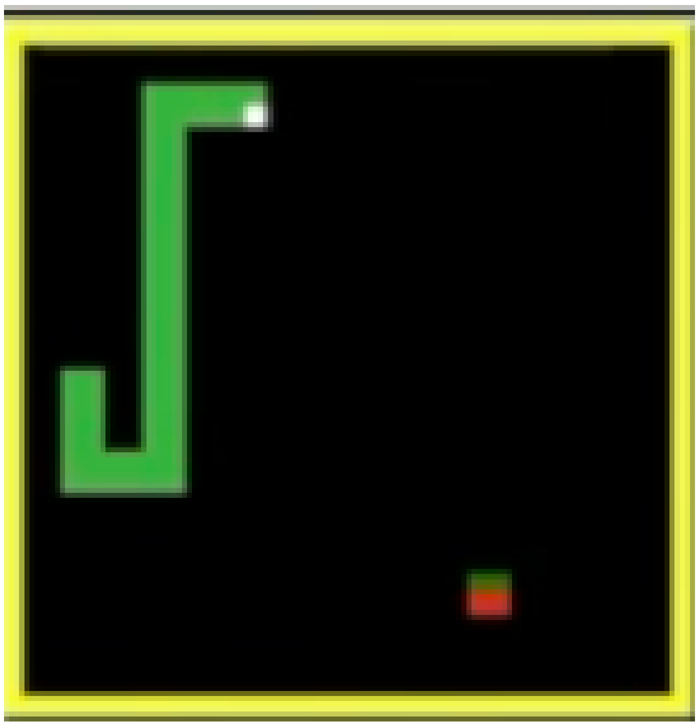

*Figure 6: View of Gameplay*