LED Pacman

Final Project Report
December 4, 2017
E155

Kai Kaneshina and Gabriel Quiroz

Abstract:

Pacman is a classic arcade game from the 1980s. Our goal was to implement a modified version of this game on a LED matrix using a Raspberry Pi and Altera Cyclone IV FPGA. The FPGA takes in user input via a keypad scanning circuit, and this information is used to control the movement of Pacman. The FPGA transmits this direction data via SPI to the Raspberry Pi, which then updates the position of Pacman. Based off of Pacman's new location, the ghost intelligently updates its movement. These changed position are then updated within a character array, which is sent back to the FPGA via SPI. The FPGA uses this data to update the LED matrix.

Introduction

Our goal was to create a simplified Pacman game, as we thought that creating a game would be fun. Furthermore, we hoped that by learning how to intelligently interpret and display a user's input, we would have the skills to recreate any game on an LED matrix.

The project consists of a Raspberry Pi and the MuddPi MarkIV FPGA board. The FPGA acts as the SPI Slave, and it is responsible for reading user input from a keypad as well as for updating a 32x32 LED matrix with information sent from the Raspberry Pi. The keypad data is encoded as a byte and then sent to the Raspberry Pi via SPI.

The Raspberry Pi acts as the SPI Master: it is responsible for keeping track of the game state and updating a 32x32 character array. Each character in the array is then converted into 3 bits of RGB color, and sent via SPI to the FPGA, which updates the matrix accordingly.

Although our game is very similar to the original game of Pacman, there are a few noticeable differences, as we did not include score, multiple ghosts, or the Power Pellets, which allow Pacman to eat the ghost. Furthermore, we implemented our own ghost AI instead of using the game's original algorithm and decided to create a new map that is different from the original.

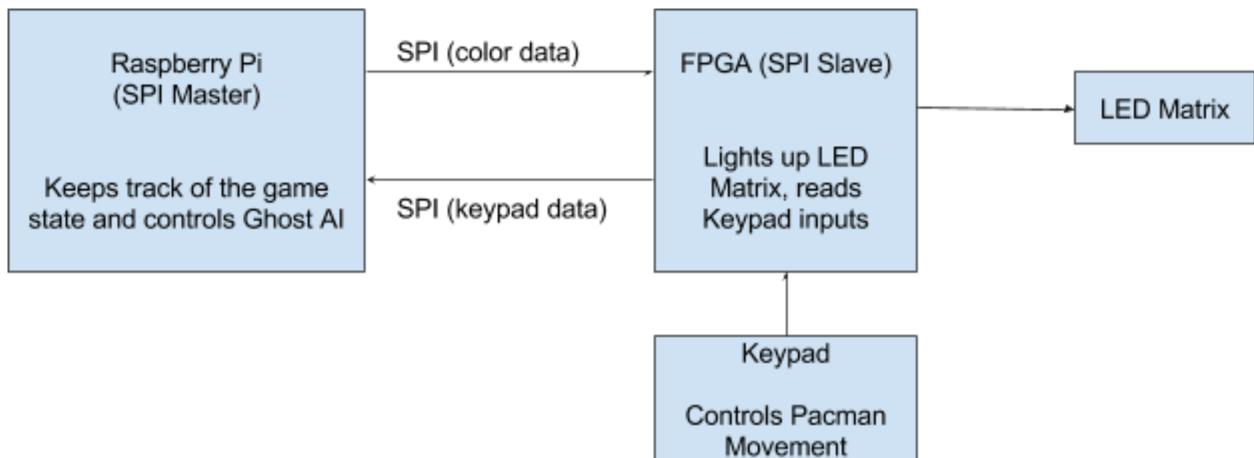

Figure 1: Basic Block Diagram of the System

New Hardware

Our new piece of hardware is a 32x32 RGB LED Matrix Panel from Adafruit. The matrix has 1024 leds that are meant to be driven by the FPGA. The back of the display contains a PCB with two IDC connectors and it requires 5V of regulated power input, which correlates to 4 amps, if all the LEDs on the display are turned on and set to white. These large power requirements will require that the device is powered with an external source. The display also requires 13 digital pins; 6 of these need to be used for bit data, and 7 which will be used for bit control.

There is not much documentation that exists for the hardware, making it difficult to understand how the matrix works. We ended up using Glen Akins tutorial to understand the matrix's internal hardware [2].

The matrix is divided into two parts, the upper and lower 16 rows. The rows are controlled with a 4 bit number called rowD, which is taken by ports A, B, C, and D on the matrix, where A is least significant bit of rowD. The matrix then decodes these 4 bits of information to figure out which two rows to drive. The rows that are driven correspond to the value of rowD and 16 + rowD. For example, to drive rows 1 and 16, rowD needs to be 0. The display is multiplexed with a 1/16 duty cycle, which means the frequency at which ROWD changes will be the frequency that our display will be multiplexed.

As for driving the matrix color, each half of the display contains a 32 shift registers for each bit of color: the top half color is controlled by R1, G1, and B1, and the bottom half is controlled by R2, G2, and B2. Thus, because there are 6 inputs of color, there are 6 separate 32 shift registers. Each register corresponds to one column of the matrix. For simplicity, we created 3 bit color buses to correspond to each half, RGB1 and RGB2. On the negative edge of each clock cycle that the matrix receives, color data is shifted into their corresponding shift registers.

We used a 5 bit counter that updated on the negative edge of the clock to keep track of which column we were updating. Once the counter overflowed, we would then increment the value of rowD, which would thus allow us to update the next row. This counter overflow would also cause us to set LAT and OE high, which allows the data to pass out of the shift registers and onto a board row, based off the value of rowD. On the negative edge of the 33rd cycle, we would then set LAT low and on the negative edge of the 34th cycle, we would set OE low. Although the tutorial described LAT as an enable signal and OE as a blanking signal, we are not entirely sure how LAT and OE work, and why they must be set in this order.

As mentioned above, we can only light up 2 rows at a time, and thus have to use time multiplexing to light up the display. Adkins' tutorial recommends performing the entire process approximately 100-200 times per second, in order to prevent flickering.

Schematics

The schematic of our entire system is shown below:

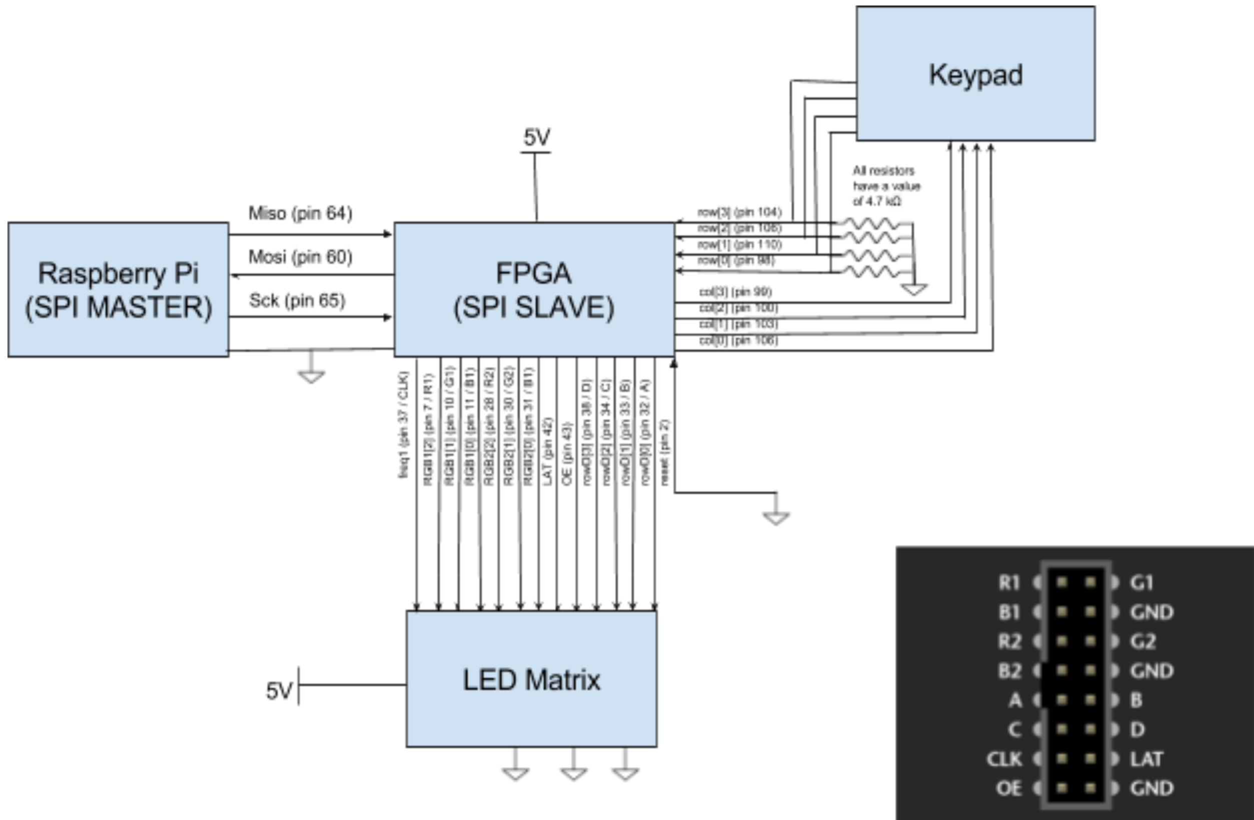

Figure 2: Schematic of LED Pacman

As stated previously, the Raspberry Pi and FPGA are connected via SPI, while the LED Matrix is driven by the FPGA. The FPGA is also responsible for reading in keypad data. The matrix signals are written next to the SystemVerilog variable names assigned to FPGA pins. A matrix connector is also shown for clarity.

Microcontroller Design

Our system used the Raspberry Pi to control the game logic for Pacman. We decided to represent the LED matrix as a 2D 32x32 char array on the pi. As Pacman and the ghosts moved around in the game, this matrix would change to represent the current state of the game. Once the array had been updated, the pi would use SPI communication to send this data over to the FPGA, which would then update the LED matrix. Since timing played a significant role in the project, we decided to decode the ASCII matrix array into 3 bits of RGB color and then combine the corresponding column bits from rows n and row n + 16 before sending the data over to the FPGA. This way, we could send data to both halves of the LED matrix simultaneously.

The game logic could be divided into three parts; Pacman movement logic, ghost movement logic, and life logic. We created a structure in C to contain the relevant data for both Pacman and the ghost (including current position and direction). Before we made any changes to the matrix, we wanted to first test if Pacman was still alive or not. If Pacman was alive, we would test if the direction either object was trying to go towards was a legal direction or not. If the next cell was not a wall, this is considered as a valid direction to move to and then we would update the game array with the new location of the ghost and Pacman.

Updating Pacman's direction was fairly simple; we used the data coming from the keypad to update the direction of Pacman. As for the ghost, we ended up using Spencer Michael's idea of updating a short array with a cell's distance away from Pacman. The ghost would use this matrix to determine which of its surrounding cells was closer to Pacman, and then move in that direction. This allowed the ghost to end up chasing Pacman in the map. To make the game easier, we ended up adding a randomness factor to the ghost's movements that would cause the ghost to choose a random direction every so often. The larger the factor, the randomness of the ghost's movements would decrease.

We also added lives to the game. If the ghost ended up being on top of Pacman, the lives would go down and both Pacman and the ghost would be reset to their initial positions. Once the number of lives reached zero, the pi would send a lose array over to the FPGA to indicate that the user is out of lives, and then reset the game. If the user managed to collect all of the food bits before they ran out of lives, we would send a win array to the FPGA, make the game faster, increase the randomness factor, and then start the game over again.

FPGA Design

The SystemVerilog code for the FPGA can be broken down into 3 modules: one that is responsible for controlling the information LED matrix, one that handles SPI communication, and one that is responsible for recording keypad data. All three of these modules were integrated to create the overall game.

While most of the LED matrix code has already been described above, it is important go over the device's timing. Because the matrix is made up of 32 shift registers for each bit of color, it was important to choose a clock that would only be high once SPI communication was finished, as this meant that the received color data would be completely sent and thus was accurate. Because our SPI module (code via [1]) would update the received value on each SPI clock cycle, it was imperative to wait until the 8th clock cycle before shifting the color data into our matrix, as this was when the data would be completely sent. Within the SPI module, we saw that there was a 3 bit counter to keep track of the number of SPI cycles. As a result, we decided to use this counter as the clock for our matrix; whenever the counter had a value of 7, we knew that the SPI transfer was complete.

As for the keypad module, it is an adapted version of lab 3, but rather than registering all strokes, we only registered the input if the number was a 1, 4, 7, or 5, which corresponded to the directions of left, down, right, and up respectively. This meant that if the user pressed an invalid key, the keypad scanning output would remain unchanged. These keystrokes only required 4 bits of data, and were send out via SPI to the Raspberry Pi, where they were used to update the position of Pacman.

Results

Our project was successful, as we were able to create a modified game of Pacman. In addition, we lived up to our promise of increasing the difficulty of the game once a level was won, as winning would not only increase the game speed but also increase the probability of the ghost choosing the shortest path rather than a random direction. Although in our initial project proposal, we planned to implement score on the LED matrix, this would have reduced the size of our game map. Furthermore, without the Power Pellets, score would only be based on the number of Pac-Dots consumed. As a result, we decided to not implement game score.

After some initial research, we discovered that each ghost within Pacman actually has its own unique AI. This was very intimidating, and thus our proposal indicated that we would not have this feature. However, after our Problem Presentation and some help from Spencer Michaels, we were able to implement an AI that would adjust its movement based on the position of Pacman. As mentioned above, we used a char array and updated its values with a cell's distance from pacman. This information was then used to update the ghost's direction, which in turn allowed the ghost to chase pacman.

As mentioned previously in the FPGA section, one of the biggest difficulties was timing, as the matrix required a clock that would only be high once the data was accurately sent. Fortunately, this was not a hard fix, as the SPI module already had a counter to keep track of bits sent. We also spent a lot of time trying to use FPGA RAM, as we hoped that it would prevent some of the matrix flickering that was observed when we simply ran the matrix off of the SPI counter clock. The goal was to read and write from memory at two different rates, as we believed that writing to the matrix with the FPGA 40 MHz clock would reduce this flicker, since the rows would be multiplexed at a faster rate. Although we were successfully figure out the timing for the writing part, we unable to read and write asynchronously without major compensation. Furthermore, the RAM caused more problems that it fixed, as it caused color bleeding between the rows and even more flickering. As a result, we decided to abandon this aspect of our project, and simply used the SPI counter clock. We were able to speed up the SPI transfer rate, which decreased the overall amount of flickering.

After getting one ghost AI to work, we thought gameplay would be improved by implementing multiple ghosts. However, this fix caused many more issues, as the ghosts would randomly replicate and instantly wipe out all the lives of the Pacman on contact. After several days of work, we realized that this implementation would require a severe overhaul of our entire ghost AI system, and we decided to not go forward with this idea.

Overall, our team was very happy with the results of our project. Although we were not able to resolve a slight flicker on the LED matrix or implement multiple ghosts, we believe that all our hard work paid off in the end.

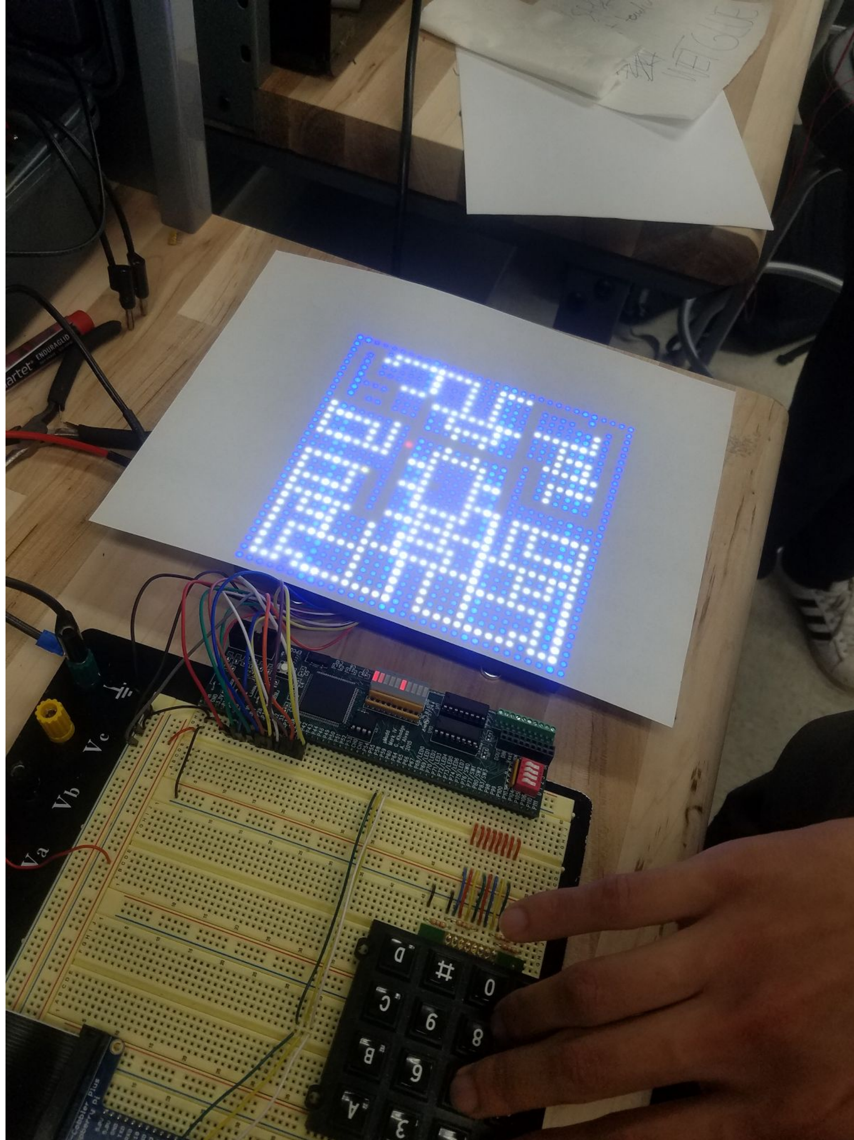

Figure 3: PacMan Gameplay

References

[1] Harris, Sarah L., and David Money Harris. "Chapter 9: I/O Systems." Digital Design and Computer Architecture, Elsevier/Morgan Kaufmann, 2016.

[2] Adkins, Glen. "RGB LED Panel Driver Tutorial." RGB LED Panel Driver Tutorial, 2014, bikerglen.com/projects/lighting/led-panel-1up/

[3] Burgess, Phillip. "32x16 And 32x32 RGB LED Matrix." Overview | 32x16 and 32x32 RGB LED Matrix | Adafruit Learning System, Adafruit, 11 Dec. 2012, learn.adafruit.com/32x16-32x32-rgb-led-matrix.

Parts

Part	Source	Vendor Part #	Price
32x32 RGB LED Matrix Panel - 5mm Pitch	Adafruit	2026	$44.95

Code Appendix

Our code can be found at the following page: https://github.com/GabeQ/E155_LED_Pacman

Keypad

5V

All resistors
have a value
of 4.7 kΩ

Miso (pin 64)

Mosi (pin 60)

Sck (pin 65)

Raspberry Pi
(SPI MASTER)

FPGA
(SPI SLAVE)

row[3] (pin 104)
row[2] (pin 106)
row[1] (pin 110)
row[0] (pin 98)

col[3] (pin 99)
col[2] (pin 100)
col[1] (pin 103)
col[0] (pin 106)

freq1 (pin 37 / CLK)
RGB1[2] (pin 7 / R1)
RGB1[1] (pin 10 / G1)
RGB1[0] (pin 11 / B1)
RGB2[2] (pin 28 / R2)
RGB2[1] (pin 30 / G2)
RGB2[0] (pin 31 / B1)
LAT (pin 42)
OE (pin 43)
rowD[3] (pin 38 / D)
rowD[2] (pin 34 / C)
rowD[1] (pin 33 / B)
rowD[0] (pin 32 / A)
reset (pin 2)

5V

LED Matrix