LED Tetris

Final Project Report
December 9, 2016
Robert Linden and Paul Slaats
GitHub Url: https://github.com/robertmlinden/e155-tetris

Abstract:

There is no easy method to create a game of Tetris with user implemented logic. To address this, a project was developed using a Raspberry Pi and MuddPi FPGA Board. To create a self reliant system with a Keypad Scanner and LED Matrix so that a system playing Tetris could function completely independent from any additional system. The final system will allow the user to strike a Keypad Scanner in accordance with pre-programmed moves that will send signals in accordance with the Keypad to the MuddPi FPGA. These signals will be sent to the Raspberry Pi via SPI Communication. A board state generated on the Pi will update in accordance with the keypad presses. This updated board state will be sent over SPI to the MuddPi FPGA which will drive the LED Matrix and produce the visual component of the Tetris system.

Introduction

We sought to build a game of Tetris displayed on a LED board. We saw a lot of potential for features that could be implemented on the FPGA and especially the Pi. The idea of implementing a game seemed fun and we liked the idea that we were working on something that any non-technical person could interface with and understand. We were able to generate a substantial list of stretch goals for when we completed the deliverables, which gave us flexibility to balance our time and enthusiasm for the project. Moreover, when we pitched the idea to Professor Harris, he was enthusiastic about the idea as it had the potential to serve as a demo in front of the MicroPs lab. This gave us a greater incentive to move forward with this project.

Our implementation of LED Tetris is split between a Raspberry Pi 3.0 and a MuddPi FPGA. All communication between the two devices is done via SPI with the Pi acting as the master and the FPGA acting as the slave. The FPGA is responsible for registering keystrokes on a 4x4 matrix keypad, encoding them as a byte, and sending this information to the Pi. The FPGA is also responsible for accepting the state of the LED board and sending digital signals to the LED matrix to make the matrix light up appropriately. All code on the FPGA is written in SystemVerilog.

The Pi encodes and updates the board state and provides the timer that governs the fall rate of the falling piece. The Pi takes the game state, lays out the appropriate 32x32 display that should appear on the LED board, and sends it to the FPGA. It accepts keystroke byte representations from the FPGA and updates the game state accordingly. All code on the Pi is written in C.

While our version of Tetris is very similar to any typical implementation of Tetris, there are a some notable rules and game dynamics:

- The user can “lead-drop” a piece, meaning that if they press the correct key, the falling piece, will instantly land on the first piece below its current location.
- In addition to the board state, the user can see the next piece, the bonus piece (if available, see next bullet), and their score.
- The user’s score increases by the number of rows eliminated on a given tick, squared.
- If a user eliminates a row and doesn’t already have a bonus piece available, a bonus piece will be generated at random. If the user presses the correct key, the next falling piece will not be the default next piece, but instead the bonus piece. This works to the user’s advantage because the bonus piece may fit better in the current board state than what would have otherwise been the next piece.
- Piece landings are “sticky” meaning that once there is a tile occupied directly below any tile belonging to the falling piece, the falling piece will be solidified and the next piece will begin falling.

Block Diagram

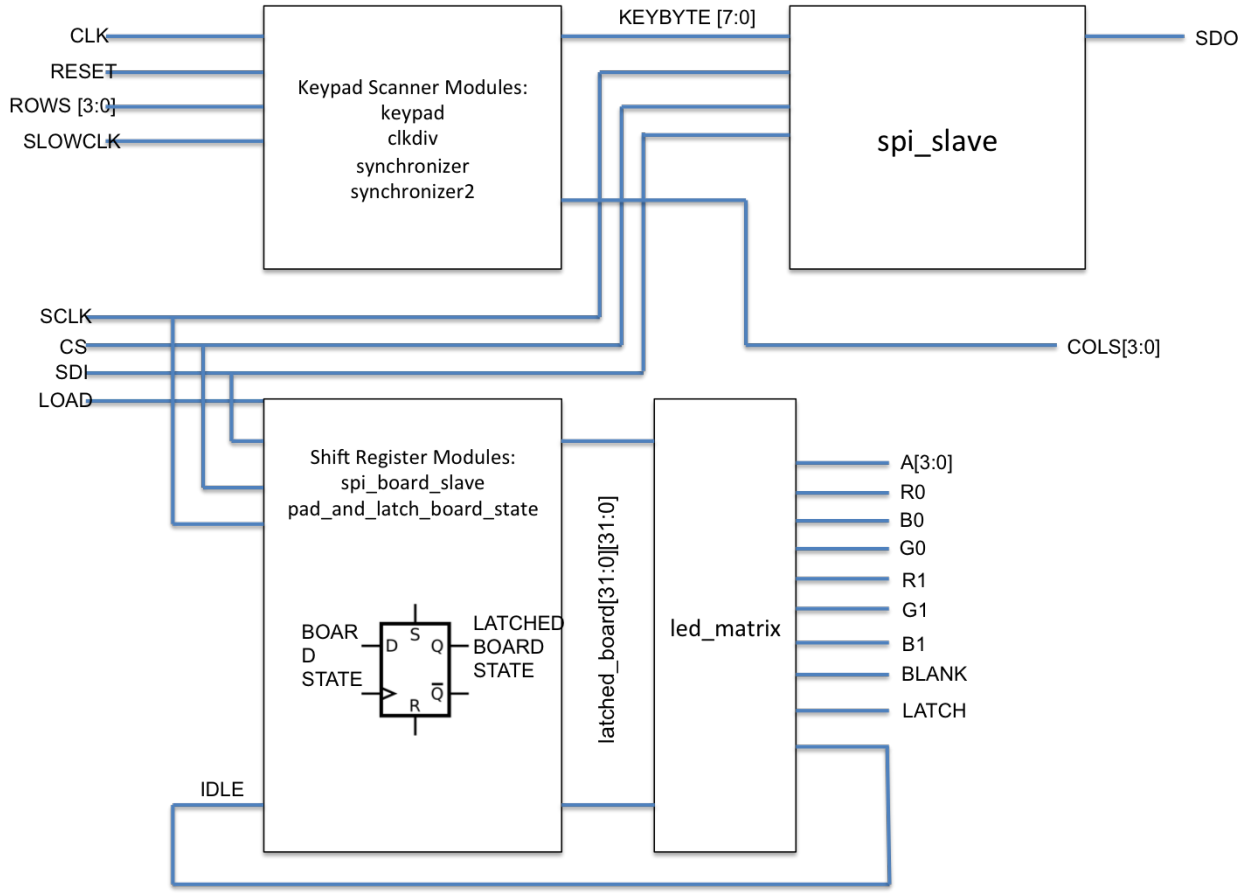

Figure 1 - Block Diagram for FPGA Logic

The block diagram in figure 1 shows the logic implemented on the FPGA. This includes the inputs to the board of CLK, Reset, Rows[3:0], SlowCLK, SCLK, CS, SDI and LOAD, as well as the outputs from the board including SDO, Cols[3:0], A[3:0], R0, G0, B0, R1, G1, B1, Blank, and Latch. This logic will conduct the necessary processing to output correct digital signals to the Pi and LED Matrix.

Schematic

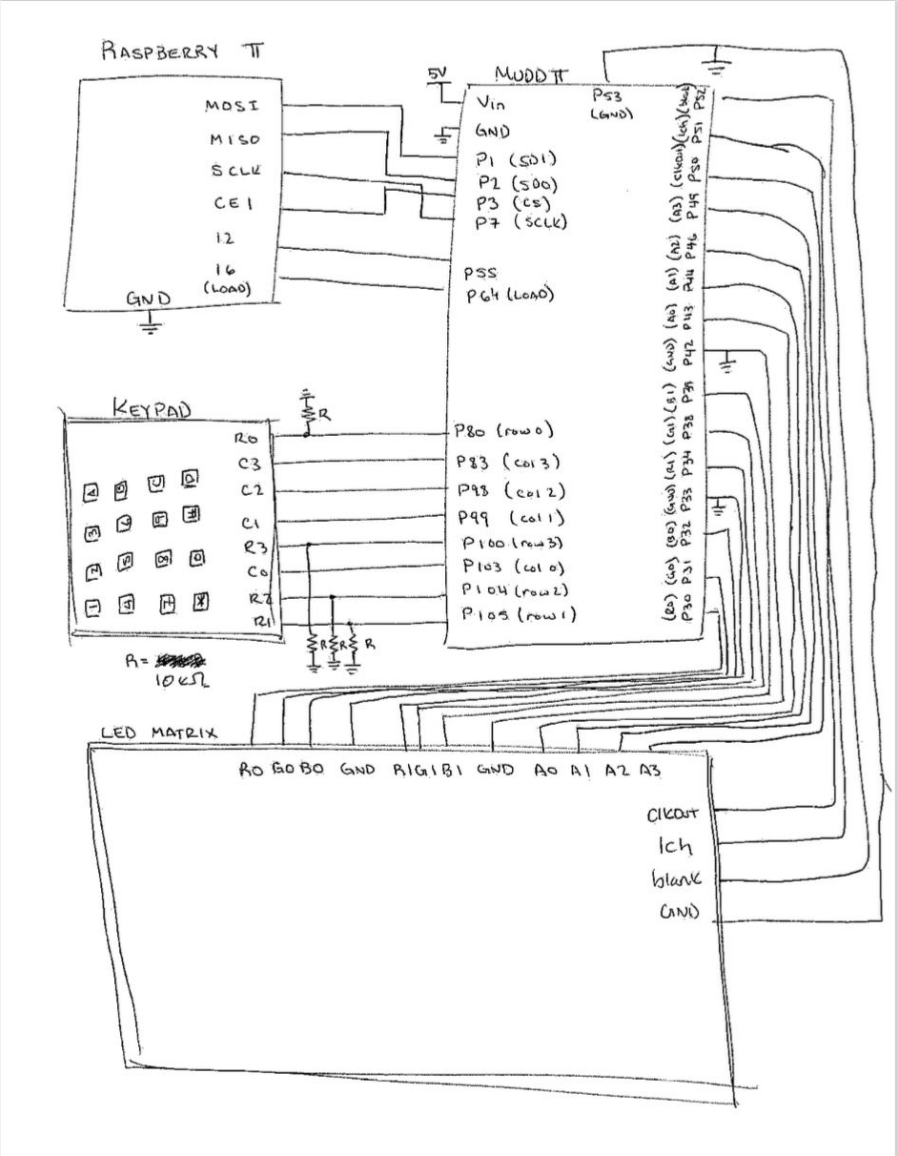

Figure 2 - Schematic of Breadboard

The schematic above depicts the circuitry implemented on our breadboard in order to establish connections between each of the elements in our system. This included connecting the Pi to the FPGA and the FPGA to the Keypad Scanner and LED Matrix.

New Hardware

We used a 32x32 RGB LED Matrix, manufactured by Adafruit. What made it particularly difficult to work with this product was that instead of providing a data sheet with timing diagrams, Adafruit instead provided a tutorial on how to drive the LED matrix using an Arduino Uno. Therefore, in order to control it with an FPGA, we had to look up resources online in which others reverse-engineered the component. This got us pretty far, but to get it working, we had to play around a little with the system ourselves.

The 32x32 matrix requires a 5V power supply and takes in 13 digital signals. The board is split into two parts, top and bottom, that are controlled independently but concurrently by two sets of RGB pins. R0, G0, and B0 control the color of the top half of the board, and R1, G1, and B1, control the bottom half of the board. Four pins, [3:0] A, are used to indicate that row A and row A + 16 are currently under control. More specifically, the LED board is comprised of 32 32-bit shift registers, one for each row. On every SCLK positive edge, the RGB bit values in row A and A + 16 are shifted by one register from column 31 towards column 0. That means that the next RGB values will first be shifted into the column 31 shift register.

After 32 clock cycles, the RGB data must be latched, or else it will simply shift out bit by bit on subsequent clock cycles. To latch the data, both the LATCH and BLANK signals should be driven high on the negative edge of the 32nd clock cycle. Then, on the negative edge of the 33rd LATCH should be driven low, and finally, on the negative edge of the 34th, BLANK should be driven low. We didn't play around with LATCH very much so we don't know exactly what it does, but resources we have found have described it as an enable signal, which must be on while the board is being latched in order for the row to light up with the RGB values shifted in.

Following the previous steps will light up a single row for a very short amount of time. In order to light up the whole board, row-multiplexing must be used. After waiting a bit of time, the value of A should be incremented to light up the next two rows and the process can be repeated exactly as above until data has been latched for every row. In order to prevent flickering, the resources we found suggested to repeat this entire process 100-200 times. We suggest that in order to ensure that all of the rows light up with the same intensity, that the row-multiplexing counter should be about 16 times slower than the counter that determines the interval between rounds of lighting up the board. That way, all of the rows are given about equal time for illumination.

Microcontroller Design

The Raspberry Pi 3.0 is in charge of two major tasks: one is updating and maintaining the game state and the other is driving the SPI communication with the FPGA. The code is loosely organized in 3 files: Tetris.c, Board.c, and Tetrislib.h. Tetrislib.h contains some constants and structs used in both Tetris.c and Board.c. Board.c contains all of the functions that update the board state such as move, rotate, tick, eliminateRow, and so on. Tetris.c contains everything else. This includes SPI functions, such as sendBoardState(), which takes the game state, converts it to the LED matrix representation and sends it to the FPGA. Tetris.c contains System Timer functions, such as delayMicrosAndWaitForKeyPress(), which runs a while loop for the duration of a game tick, and within the while loop, checks for new key presses and makes calls to update the game state accordingly. Tetris.c also contains the main method, which drives the flow of the game. Every time something interesting happens (move, rotate, game tick), processTick() is called, which delegates calls to functions in board.c to update the game state.

FPGA Design

The FPGA is comprised of four primary modules: one that registers keystrokes, one that sends them to the Pi, one that reads in the board state from the Pi, and one that controls the LED board. The former two and the later two operate independently from one another.

The module that registers keystrokes is adapted from Lab 3. The FPGA registers a keystroke from when the key is first pressed until it is released. at which point the keypad module defaults to 'D', which is not used for gameplay. Before passing this information to the Pi, the key pressed is encoded as a byte of the format X000####. X is 1 if a key was pressed and 0 otherwise, and #### is the binary representation of the key just pressed. The spi_slave module is taken from Prof. Harris' *Digital Design and Computer Architecture*. Its purpose is to send data just from the FPGA to the Pi; the byte sent from the Pi to the FPGA during the keyByte transfer is junk and is just meant to initiate the SPI transfer.

Another SPI module "spi_board_slave" takes in 1024 bytes in sequence and populates a 32x32 matrix representing the LED board with the incoming values. The module resets its matrow and matcol counters at 0 so long as the LOAD signal is low. Once the LOAD signal goes high, the counters increment appropriately with the serial clock to store the led matrix data in the right array slot.[1] Once the entire board has been read in through SPI, we latch the board and display it on the LED board. This is done in the the led_matrix module, which is described in more detail in the "New Hardware" section above.

[1] Actually, as the code stands as of 12/09/2016, we only store one *bit* per pixel due to logic gate limitations on the FPGA. This is explained in more depth in the "Results" section just below

Results

Our hard work paid off and the project was a success! We even surpassed our deliverables, adding lead-dropping and score display. Looking back, the most difficult aspects of our project were the SPI for the board state and storing the board state with the limited FPGA logic gates.

For SPI, we first ran into issues with chip enable. When we read the signal with an oscilloscope, we noticed that the signal was spiking abruptly rather than going high, staying high for about 8 sclk cycles, and dropping down low again. We were able to avoid using chip enable by relying on our LOAD GPIO signal, which stayed high only during the SPI transfer. We effectively used it as a reset for the row and column counters, used to keep track of which row and column the incoming byte representing a single LED belonged to. The second design challenge we faced with SPI was deciding on how to encode the board for sending over SPI and how to properly receive it on the FPGA, maintaining the correct structure of the board. Originally, we planned to send the game board, next piece, bonus piece, and score via SPI separately, but it turned out to be easier to send the LED board state. It left much less work for the FPGA, as the FPGA didn't have to piece together the game state into the led board state; it could just read the led board state, store it, and iteratively send digital signals to the LED matrix to display the image. In fact, we essentially wrote a generic library on the FPGA to receive a 32x32 "black-and-white image" and display it on the LED matrix. The FPGA isn't even aware that Tetris is being played! This implies that the FPGA code could be reused to develop any visual project driven by logic on the Pi.

However, with this design decision, issues of time and space arise. Sending 8 bits of data for 1024 LED pixels with a 1 MHz serial clock takes almost a tenth of a second (while we could have sent 1 bit of data for every pixel to optimize our current protocol, we decided to keep 8 bits for future work that includes pixel color). Moreover, representing 1024 bits of data and permitting asynchronous reads takes up half of the logic gates on the FPGA; so much so, in fact, that it was too difficult to try to squeeze 2 bits per pixel on the FPGA to add 2 more colors to Tetris.

There are workarounds for both issues. To decrease the amount of total time spent on SPI, it is not possible to initiate SPI transfers less frequently, because then the board state displayed on the LED matrix would not update in real time. Instead, there are two options. One would be to implement SPI using a faster clock; for example, the FPGA's 40 MHz clock. Another way would be to decrease the size in bytes of the transfer by only sending over pixels that *have* changed. This would require pixel coordinates to be send with each pixel in addition to its new color, so this solution would work best when there are very few pixels changing each time the display changes.

The only way avoid the issue of limited space on the FPGA is to synthesize RAM. Once this is accomplished, the amount of data per pixel able to be stored is practically limitless. While implementing a RAM module is fairly easy (there are many sources), using it is hard, as now

inputs and outputs to the module must be juggled around the various modules that need to access the RAM, as asynchronous reads are no longer an option.

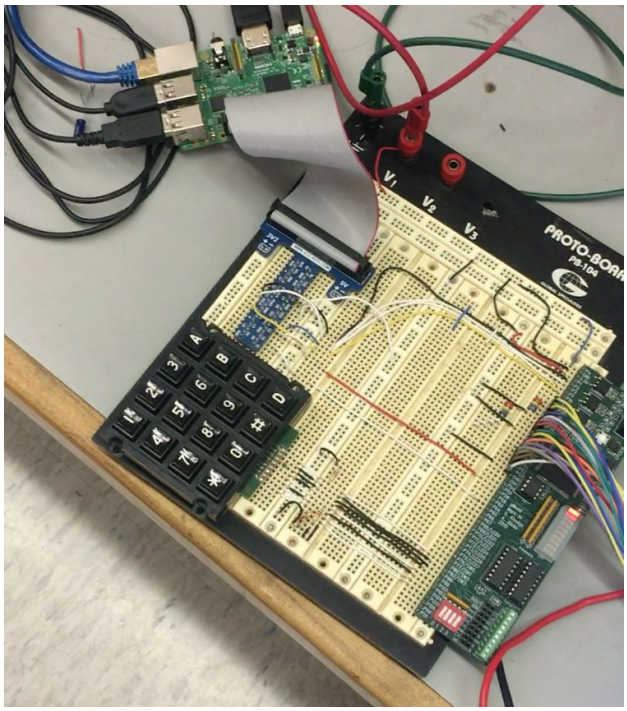

Figure 3 - Breadboarded System

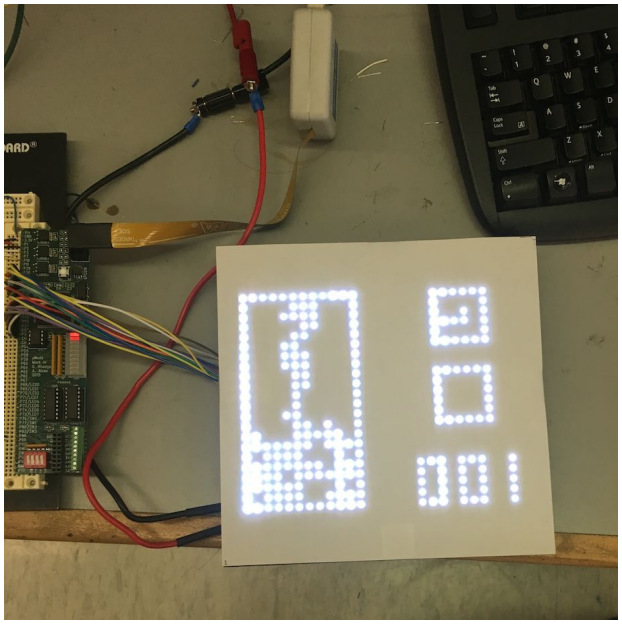

Figure 4 - Unskilled Tetris Gameplay

References

"32x32 RGB LED Matrix." *Overview | 32x16 and 32x32 RGB LED Matrix | Adafruit Learning System*. N.p., n.d. Web. 09 Dec. 2016. <https://learn.adafruit.com/32x16-32x32-rgb-led-matrix/>.

Harris, David Money., and Sarah L. Harris. *Digital Design and Computer Architecture*. 2nd ed. Amsterdam: Morgan Kaufmann, 2013. Print.

"RGB LED Panel Driver Tutorial." *RGB LED Panel Driver Tutorial*. N.p., n.d. Web. 09 Dec. 2016. <http://bikerglen.com/projects/lighting/led-panel-1up/>.

Parts List

Part	Source	Vendor Part #	Price
32x32 LED Matrix	Adafruit	2026	$40.00

Appendix: Code

Check out our github code at https://github.com/robertmlinden/e155-tetris. Relevant files are in the uppermost directory. See the “Microcontroller Design” section above to see how the Pi code is split up between the 3 C files.