E155 Microprocessors

Final Report

Author

Yoni Maltsman Joseph Han

Professor

Joshua Brake

December 10, 2021

Contents

1 Abstract

We interface a PS/2 keyboard with addressable WS2812B LED strips to create custom lighting patterns that correspond to key presses. We use a Field-programmable Gate Array(FPGA) to capture keypress signals and drive LED displays, maximizing the ability of the FPGA to parallelize processes and freeing up resources for the Microcontroller Unit(MCU) to handle the processing of keypress data and the generation of lighting patterns. Data transmits between the FPGA and the MCU over SPI and the keyboard uses the PS/2 protocol and the LED strips use the non-return-to-zero protocol, two new additional protocols to this project, for which specialized modules are implemented in the FPGA.

2 Introduction

2.1 Motivation and Overview

There are gaming keyboards on the mass market with LED back lightings. However, the lighting often comes in a fixed or predesigned pattern. Our goal was to interface a keyboard with LEDs through an FPGA and microcontroller, so that one can generate customized patterns on the LEDs as they're typing. The patterns can be hard-coded in C and uploaded to the MCU, allowing some degree of customizability. In our implementation, the LEDs are physically separate from the keyboard, but this can serve as a prototype for a more integrated product where the LEDs are built into the keyboard, and a user can generate interesting patterns as they're typing away.

In order to achieve this goal, we design mechanisms to sample the keyboard when a user presses a key and continuously generate and send patterns to the LEDs. Putting these two mechanisms together poses multiple challenges especially in terms of potential latency and responsiveness of the pattern to keypresses, and we maximized the ability of the FPGA to parallelize processes to sample the keyboard inputs and drive the LEDs, while the MCU focuses solely on processes the keypress data and generating the pattern, allowing flexibility in hardcoding the pattern from the MCU side.

2.2 Block Diagram

Figure 1 and Figure 2 show the overall and detailed block digrams for both the FPGA and MCU and how individual submodules are connected to each other.

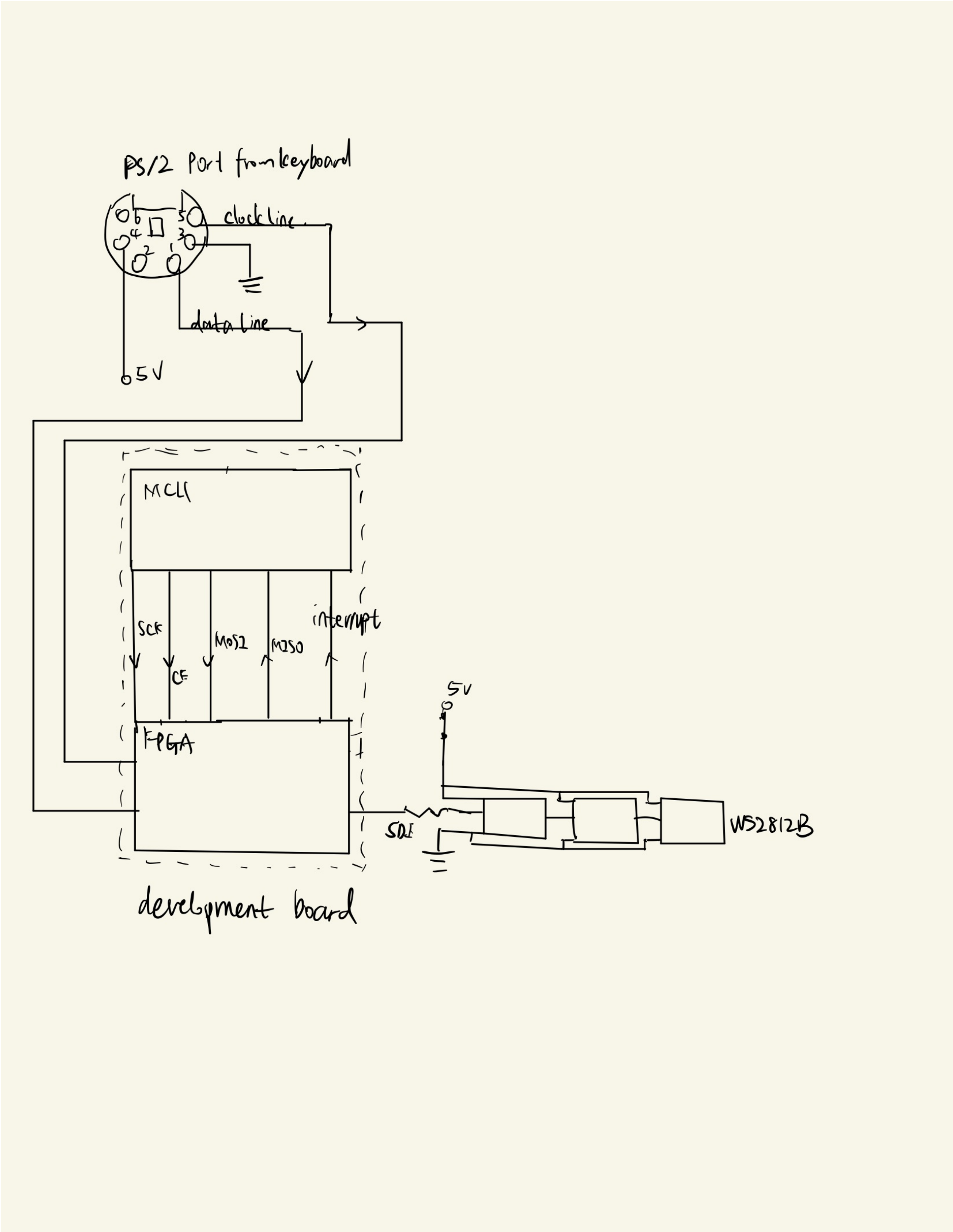


Figure 1: Overall Block Diagram

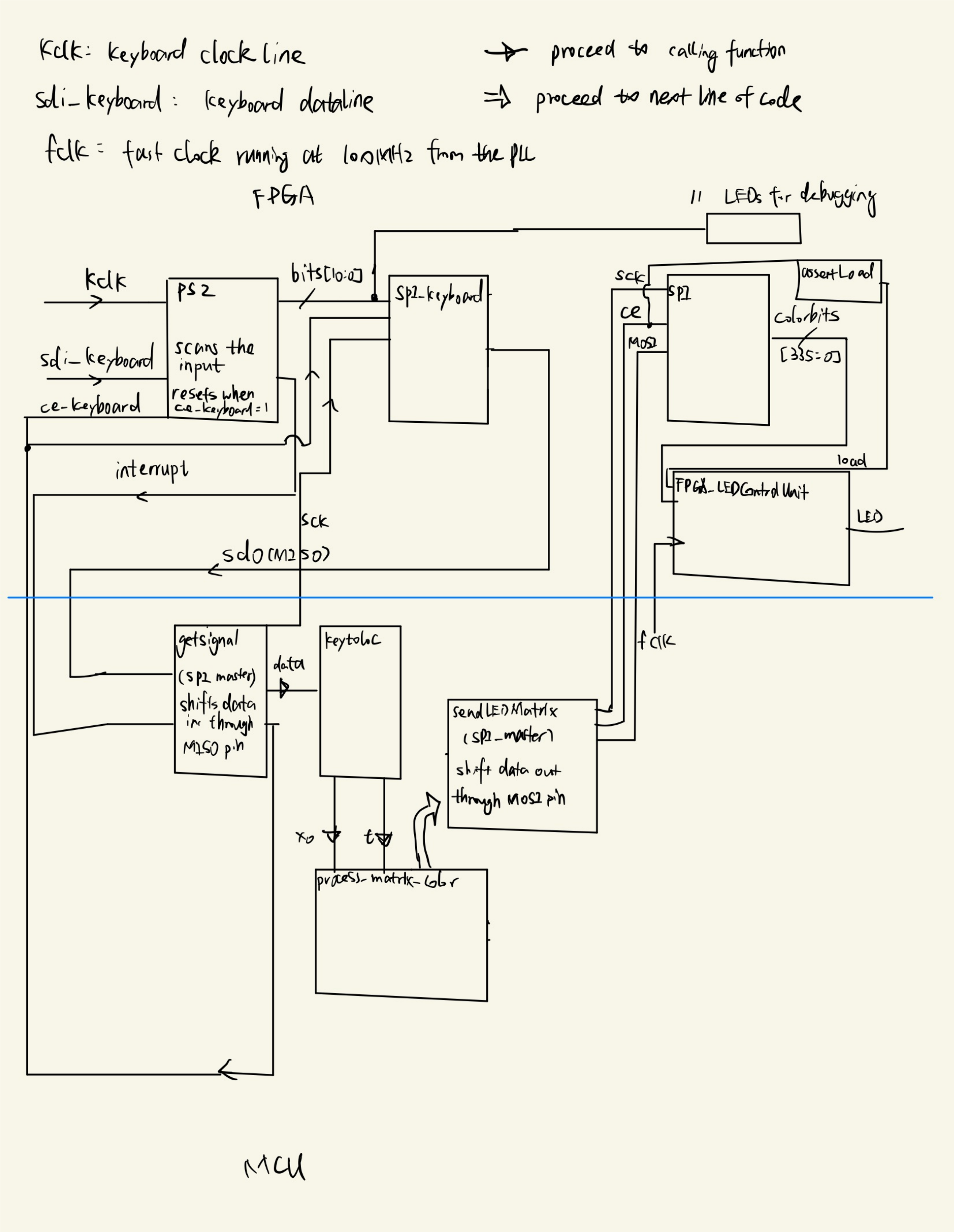


Figure 2: Detailed Block Diagram

3 New Hardware

3.1 PS2 Keyboard

We used a keyboard that uses the PS/2 protocol. Details of the PS2 protocol are as follows:

- The keyboard must be connected to 5V power, and has a data line and clock line which must be pulled up to power.
- Check on the oscilloscope to ensure that the highs and lows for the clock line and data line both fall within the logic levels of the FPGA to properly capture shifts in clock signal as well as data signals. Adjust pull-up resistance values accordingly.
- When powered and idle both the clock line and data line should be high.
- When a key is pressed, the keyboard generates 11 clock cycles on the clock line, during which it sends 11 bits of data along the data line. The first bit is a start bit, which must always be zero, which is followed by a byte of data corresponding to which key was pressed. It concludes with a parity bit and a stop bit.
- The data signal stays constant on the negative clock signal, allow sampling both on the negative clock edge, and throughout the negative clock signal. (Note on figure 3 that the width of the each data signal is larger than the negative clock signal, allowing sampling throughout the negative clock).

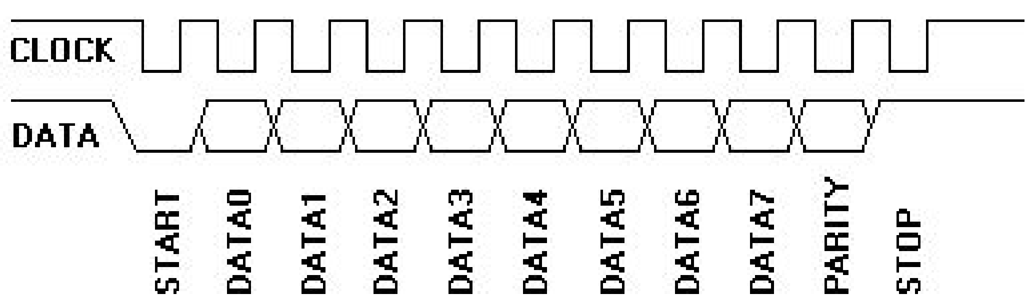


Figure 3: Transmission over the clock and data lines for the PS/2 protocol

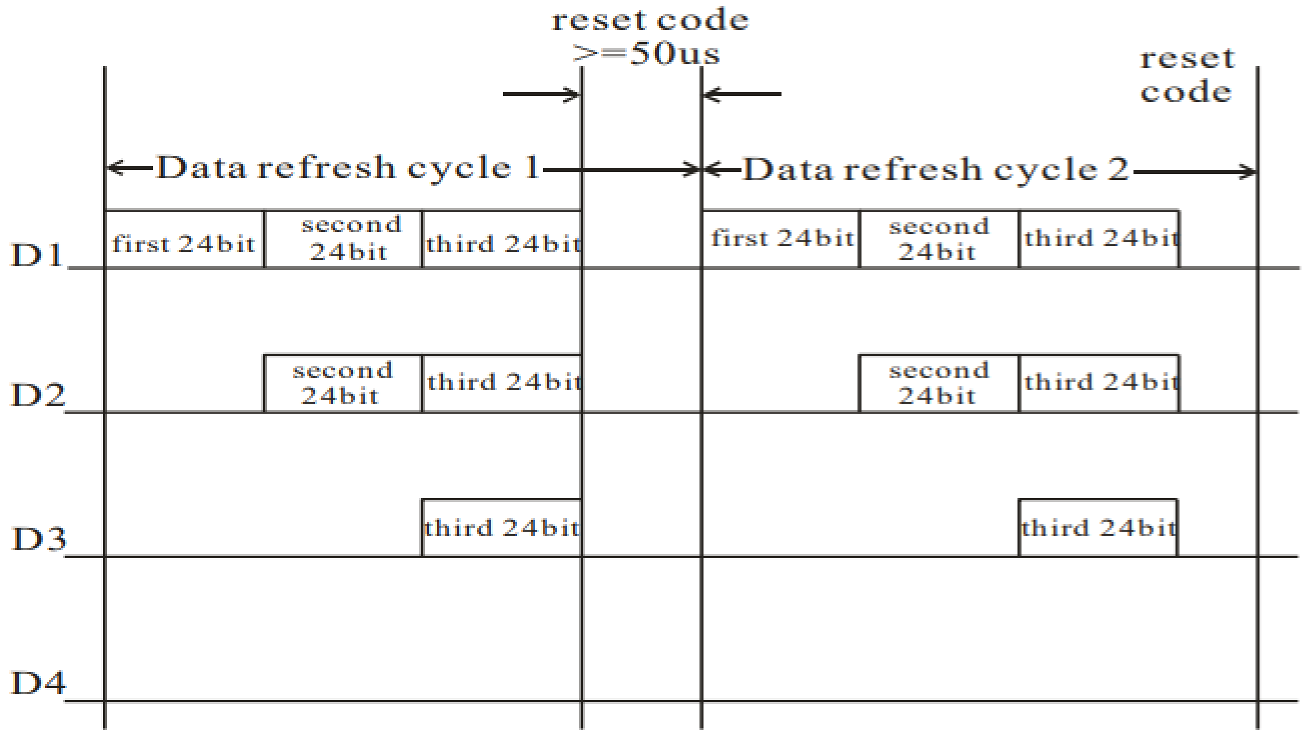


Figure 4: Latching and transmission of data in LED pixels

3.2 WS2182b LEDs (Neopixels)

- An LED strip consists of pixels, which are integrated circuits each with their RGB LEDs.
- The data lines are wired in series and power and ground in parallel.
- Each chip in a pixel is connected to power and ground, and has an input line for receiving data and an output line for transmitting data to the following chip. Thus, color data must be transmitted sequentially for a given strip. For example, if one wanted to light up six pixels with six different colors, one would have to send the data for all six colors sequentially to the data line of the first pixel. This pixel would latch on to the first color, and then pass the next five colors to the second pixel (Figure 4).
- The chip for each pixel drives 3 onboard Pulse Width Modulation(PWM) modules that determines the intensity of the Red, Green and Blue colors of the LEDs.
- Color is represented by 24 bits, with 8 bits encoding the intensity of green, red, and blue, in that order. For example, the hexadecimal representation of yellow is 0xFFFF00. However, one does not simply send these bits to the LEDs but has to encode them further according to the NZR communication mode. 1's and 0's are encoded as high and low signals with specified durations (Figure 5).

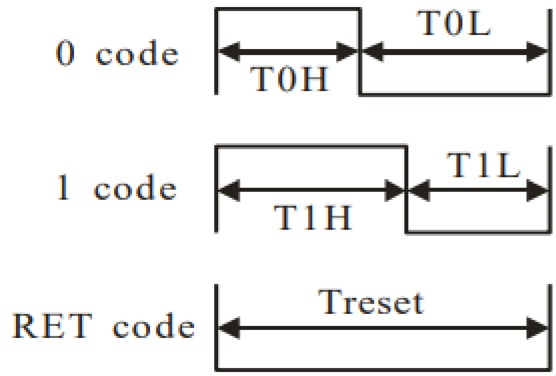


Figure 5: The NZR Communication Mode: Signals for 1 and 0 codes on the LEDs

4 Microcontroller Design

We utilize 'bare-metal' programming to implement our project. That is, following initialization the MCU enters an infinite while loop, during which it processes keyboard data if a key has been pressed, and sends either a plain or patterned LED array to the FPGA over SPI depending on whether a key has been pressed and the time that has elapsed since the press (Figure 6).
At the beginning of the while loop, the MCU checks for a keypress. The keypress is first sampled by the FPGA, after which it raises an interrupt flag. The MCU detects the keypress by MCU checking for the interrupt flag from the FPGA, upon which it intiates an SPI transaction to receive the parsed data from the FPGA. This is a faster scheme than having the MCU sample the keyboard inputs, as the FPGA, with a fast PLL clock running at 100MHz, can pick up a keypress almost instantly.
The MCU generates a wave pattern by assigning a value to each pixel based on a wave function that takes a time counter t and location of keypress x_0 as inputs. When the MCU processes a keypress, it initiates a counter t. It also maps the key data that it received to a location x_0. t and x_0 are fed as inputs to a function that generates a pattern. The pattern is implemented as a travelling wave which begins at the location of the keypress. A pixel at location x on the LED matrix will get the following intensity at time t after a keypress at x_0:

$$f(x,t) = e^{\frac{-(x-ct-x_0)^2}{2w^2}}$$

Where w is the width of the travelling wave, and c is its speed.
The travelling wave equation gives the pattern a more continuous, natural look. If no key is pressed or the counter reaches its limit, the MCU simply sends a plain array to the FPGA.

5 FPGA Design

The FPGA

- samples the keypress using an FSM,
- sends it to the MCU over the SPI MISO pin,
- receives color data from the MCU over the SPI MOSI pin,
- converts it to signals that are compatible with the Neopixels using an FSM generating the PWM wave forms corresponding to the NZR communication mode.

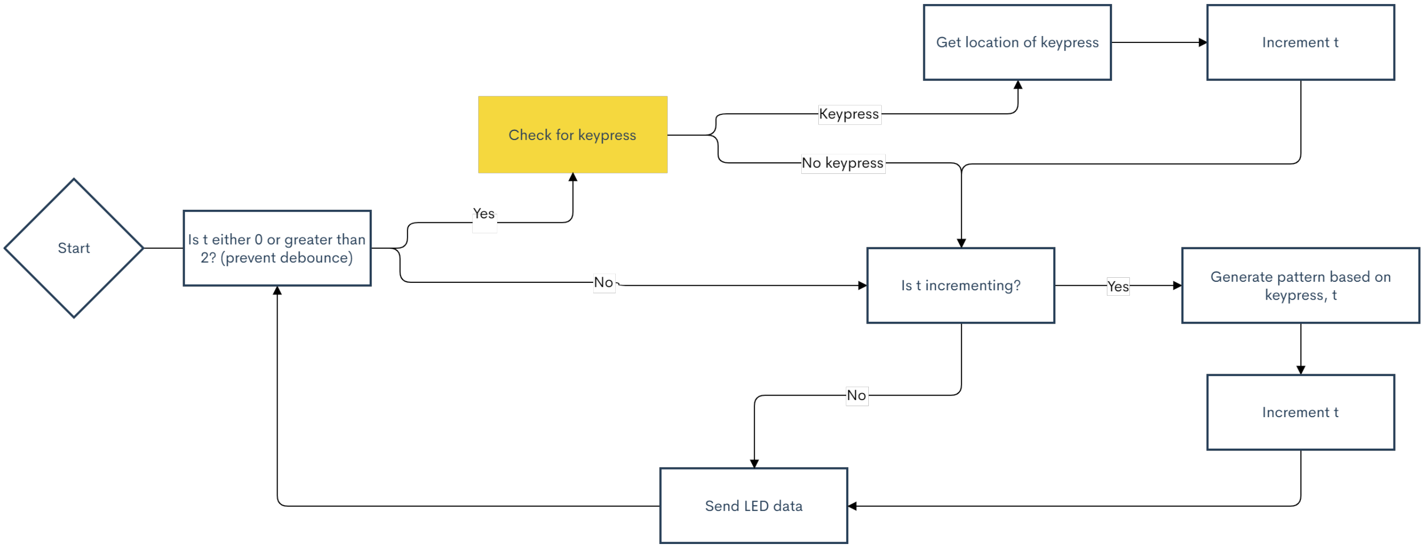


Figure 6: MCU while loop for sampling the keyboard and sending the LED array.

The state transition diagrams for the two FSMs used for sampling keyboard inputs and generating waves to drive the LEDs are shown in Figure 7 and Figure 8.

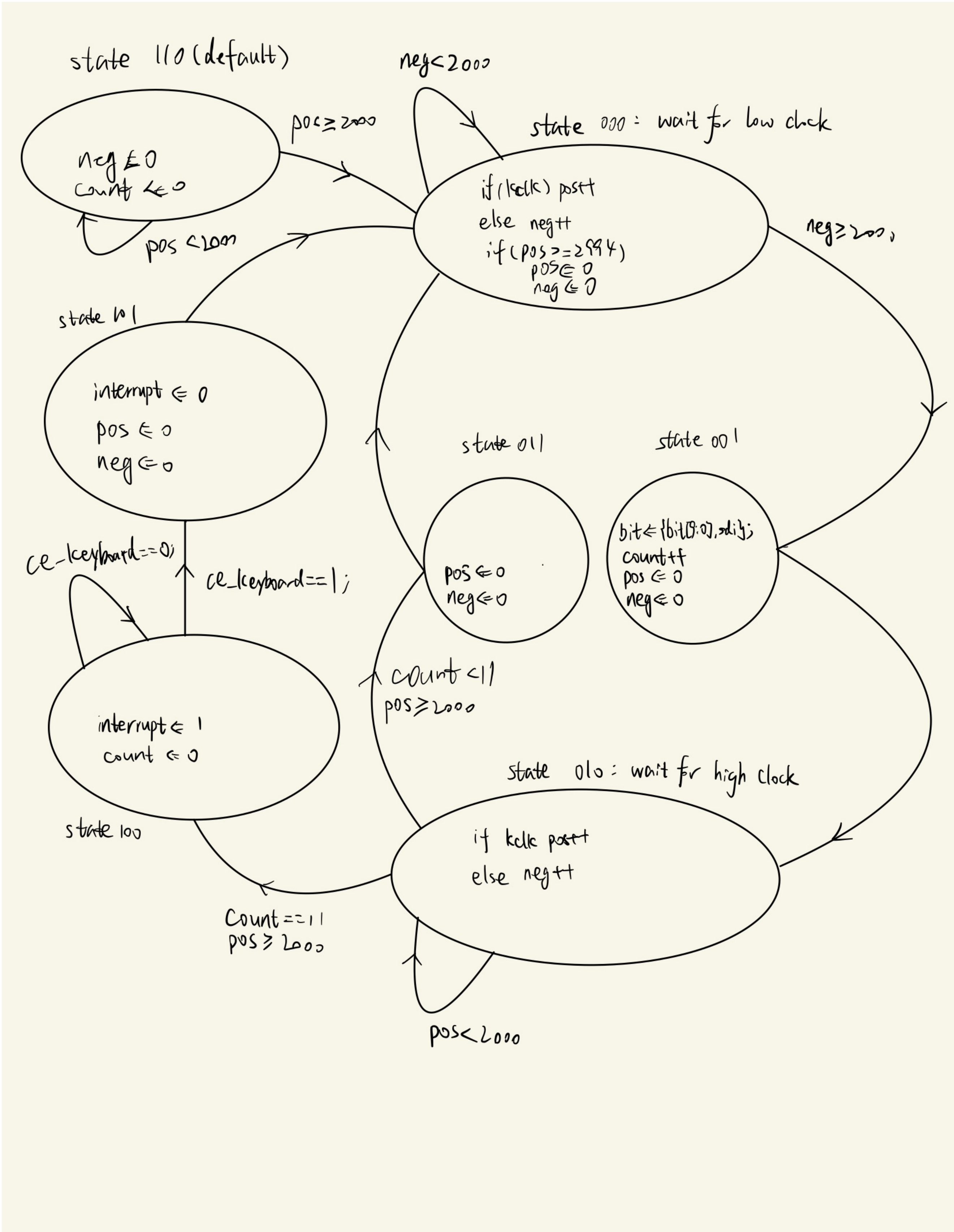


Figure 7: FPGA finite state machine for capturing key presses

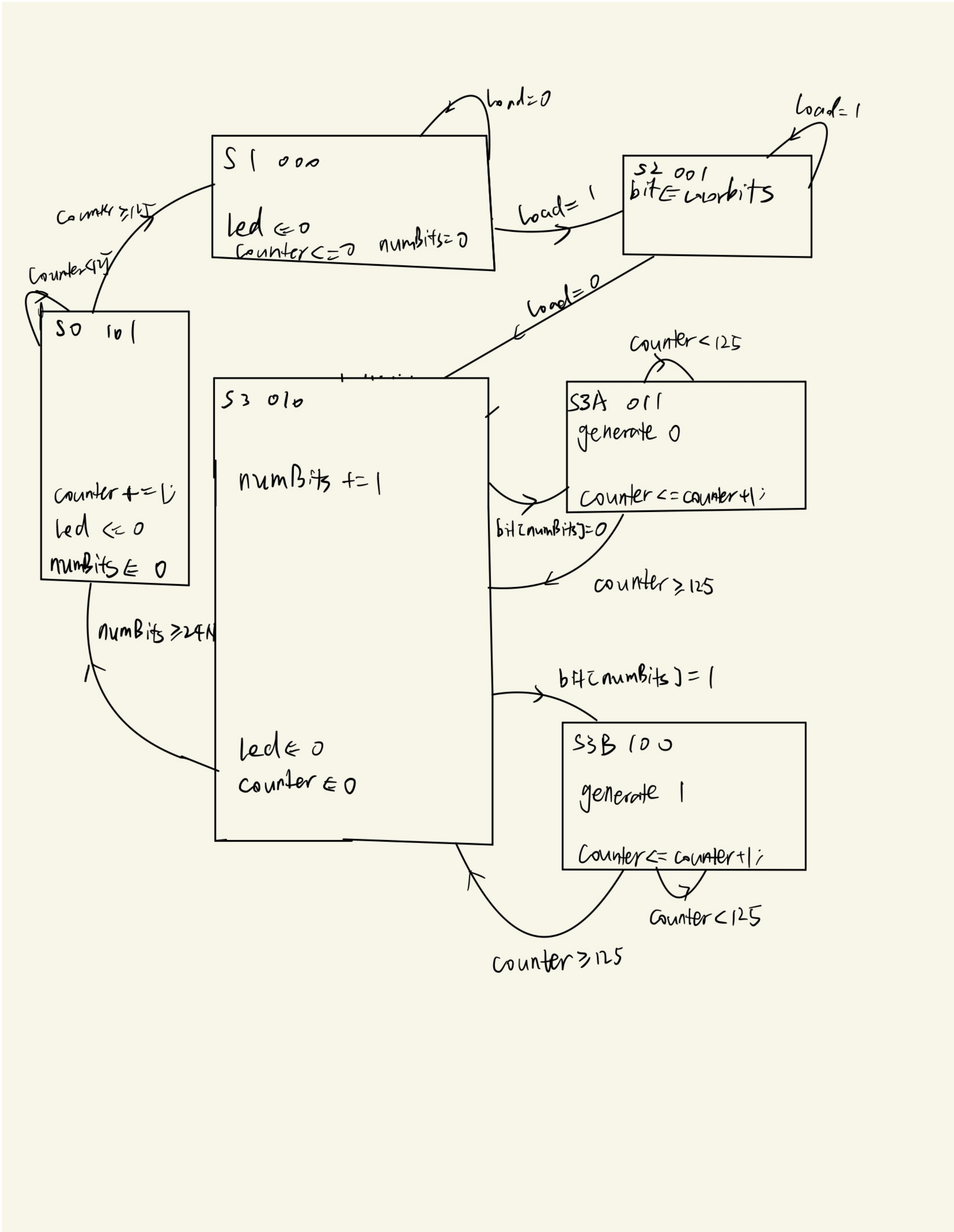


Figure 8: FPGA finite state machine for capturing key presses

6 Results and Discussion

We were able to achieve the following functionality:

- The FPGA captures a keypress
- The FPGA decodes the keyboard data signal
- The FPGA sends the data signal to the MCU over SPI
- The MCU decodes the data signal
- The MCU substitutes appropriate location x and timestep t parameters in the wave equation for each pixel for each iteration of the while loop
- The MCU sends an array of colors to the FPGA for each iteration of the while loop
- The FPGA receives the color arrays over SPI, encodes them into NZR signals and drives the LED display.

We tested out the design on a 2×7 LED strip and the wave generated travels from the start of the first LED strip to the end of the second LED strip. The LEDs show a randomly generated color at the crest of the travelling wave. For future expansion on the project, a dictionary can be implemented on the MCU to faster map data received from the FPGA to a specific key (e.g. ”A” or ”Num Lock”), and a distance helper function can be implemented to calculate the euclidean distance of keys across different rows to enable an expansion of the wave equation into 2D, allowing a rippling visual effect.

7 References

PS/2 protocol: https://www.avrfreaks.net/sites/default/files/PS2%20Keyboard.pdf
WS2182b LEDs datasheet: http://cdn.sparkfun.com/datasheets/BreakoutBoards/WS2812B.pdf

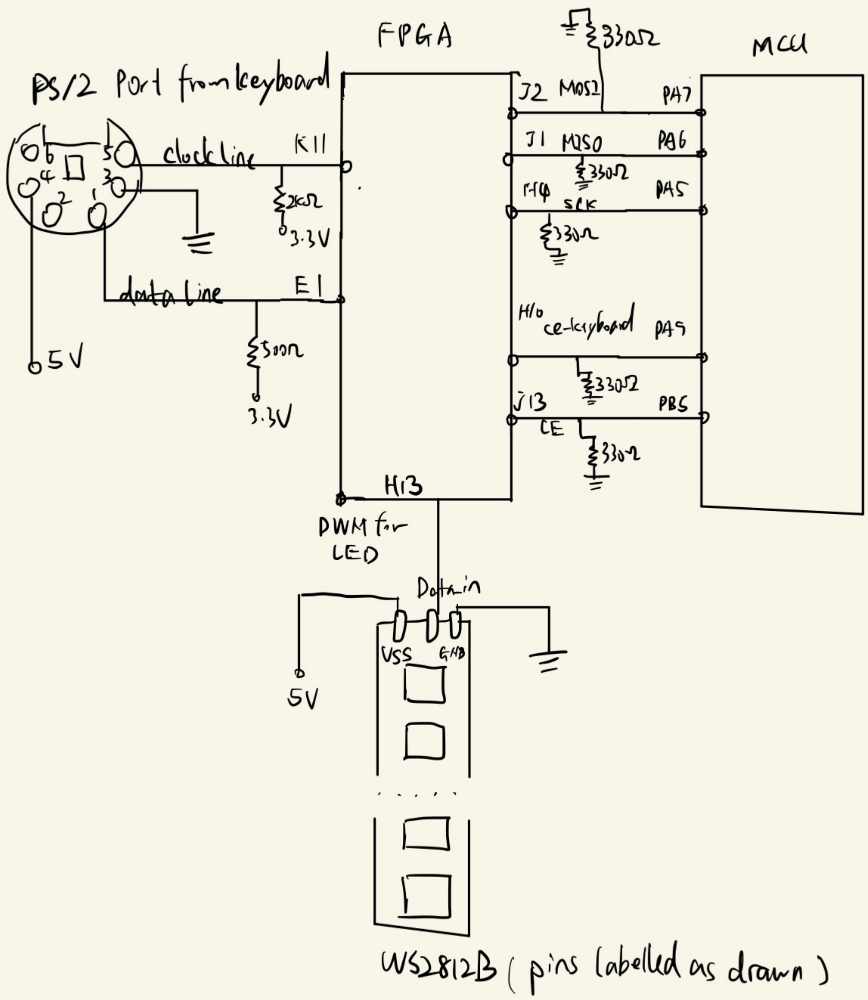
8 Bill of Materials

- PS/2 Keyboard: from stockroom
- PS/2 wired connector: https://www.adafruit.com/product/804
- WS2182b LEDs: https://www.amazon.com/ALITOVE-Individually-Addressable-Programmable-Waterproof/dp/B019DYZ
- Resistors: 1 2k Ohm, 1 500 Ohm, 5 330 Ohm
- STM32F401RE Microcontroller: Supplied for E155
- MAX 1000 FPGA: Supplied for E155
- μmudd shield, breadboard adapter, breadboard: Supplied for E155
- HP 6236B Power Supply: HMC digital lab

PS2 wired connector: WS2182b LEDs:

9 Appendices

9.1 Appendix A: Breadboard Schematics



9.2 Appendix B: MCU code

main.h: macros

```
// main.h



#ifndef MAIN_H
#define MAIN_H

#include "STM32F401RE.h"

////////////////////////////////////////////////////////////////////////////////
// Custom defines
////////////////////////////////////////////////////////////////////////////////


#define _USE_MATH_DEFINES
#define M 2 //number of strips
#define K 2 //number of keys
#define LOAD_PIN 5 //PB5, CE for sending LED data
#define K_CLK 0  //PA0, keyboard clock input for I1
#define K_DATA 0 // PB0, keyboard data input for I1
#define Ready_PIN 9 //PA9, CE for FPGA to send keypress over SPI
#define FPGA_FLAG 8 //PA8, flag from FPGA when it sends interrupt

#define SUCCESS_LED 4 //for testing

//keypress data struct
typedef struct {
  union {
    struct {
      unsigned int start : 1;
      unsigned int data : 8;
      unsigned int parity : 1;
      unsigned int stop : 1;
    };
    int raw;
  };
} ps2_frame_t;

//key struct - data and corresponding LED matrix location for a key
typedef struct {
  unsigned int data : 8;
  unsigned int loc  : 8;
} key;
```

```
//press struct - contains location and time elapsed for a press
typedef struct {
  unsigned int loc   : 8;
  float t;
} press;



#endif // MAIN_H
```

altcolorpattern2D.c

```
#include <stdio.h>
#include <math.h>
#include "STM32F401RE.h" //https://github.com/joshbrake/E155_FA2021/tree/main/labs
#include "main.h"

//////////////////////////////////////////////////
// Constants
//////////////////////////////////////////////////

#define N 7 //pixels in a strip

#define BASE_COL 255 //base color

//function declarations
void init_2DLED(uint8_t LED[M][N][3], uint8_t color[3]);
void sendLEDarray(uint8_t LED[N][3]);
void sendLEDmatrix(uint8_t LED[M][N][3]);
void process_matrix(uint8_t LED[M][N][3], uint8_t LED0[M][N][3], int x, float t);
double wave_function(int x, int x0, float t, float c);
int keytoloc(key keys[K], uint8_t data);
uint8_t getkey();
void init_keys(key keys[K]);
void getrand(uint8_t color[3]);
process_matrix_color(uint8_t color[3], uint8_t LED[M][N][3], uint8_t LED0[M][N][3], int x0,fl
uint8_t getsignal();


int main(void){
    // Configure flash latency and set clock to run at 84 MHz
  configureFlash();
  configureClock();
```

```
//initialize list of keys with their mappings
key keys[K];
init_keys(keys);

// Enable GPIOA clock
RCC->AHB1ENR.GPIOAEN = 1;
RCC->AHB1ENR.GPIOCEN = 1;

// "clock divide" = master clock frequency / desired baud rate
// the phase for the SPI clock is 1 and the polarity is 0
spiInit(1, 0, 0);

//configure pins
pinMode(GPIOB, LOAD_PIN, GPIO_OUTPUT);
pinMode(GPIOA, FPGA_FLAG, GPIO_INPUT);
pinMode(GPIOA, Ready_PIN, GPIO_OUTPUT);
digitalWrite(GPIOA, Ready_PIN, 0);


//initialize LED matrix
uint8_t LED0[M][N][3];
uint8_t LED[M][N][3];
uint8_t color[3] = {0xFF, 0xFF, 0xFF};

init_2DLED(LED0, color);
init_2DLED(LED, color);
sendLEDmatrix(LED);
uint8_t data;


//variables for pattern
float t = 0;
int x0;
float dt = 0.01;
float end = 500;

while(1){
    if (t == 0 || t > 2){
        if (digitalRead(GPIOA, FPGA_FLAG)){ //check for FPGA flag
            data = getsignal(); //SPI transaction for key data
            x0 = keytoloc(keys, data); //map key to an LED location
            t = .01;
            getrand(color);
        }
    }
```

```
/*
    if (t < .01){
        sendLEDmatrix(LED);
    }
    else if (t >= end){
        t = 0;
    }
*/
    if (t >= 0.009 && t <= end) {
        process_matrix_color(color, LED, LED0, x0, t);
        sendLEDmatrix(LED);
        t += dt;
        if (t == end){
            t = 0;
        }
    }



  }

}

uint8_t getsignal(){
    //Conduct SPI transaction with FPGA to receive keypress data
    uint8_t data;

    digitalWrite(GPIOA, Ready_PIN, 1);
    data = spiSendReceive(0);
    while(SPI1->SR.BSY);
    digitalWrite(GPIOA, Ready_PIN, 0);
    //data = data << 1; //left shift by 1 to make up for weird right shifting
    return data;
}

process_matrix_color(uint8_t color[3], uint8_t LED[M][N][3],uint8_t LED0[M][N][3],int x0,floa
    /*
    Applies a pattern to the LED matrix, where the pattern is a single travelling wave (with
    */
    int i;
    int j;
    int x;
    float c = 3;
    for (j = 0; j < M; j++){
        for (i = 0; i < N; i++){
```

```
            x = i + N*j;
            LED[j][i][0] = 2*LED0[j][i][0] - wave_function(x, x0, t, c)*color[0];
            LED[j][i][1] = 2*LED0[j][i][1] - wave_function(x, x0, t, c)*color[1];
            LED[j][i][2] = 2*LED0[j][i][2] - wave_function(x, x0, t, c)*color[2];

        }
    }

}


void process_matrix(uint8_t LED[M][N][3], uint8_t LED0[M][N][3], int x0, float t){
    /*
    Applies a pattern to the LED matrix, where the pattern is two diverging travelling waves
    */
    int i;
    int j;
    int x;
    for (j = 0; j < M; j++){
        for (i = 0; i < N; i++){
            x = i + N*j;
            LED[j][i][0] = LED0[j][i][0] - wave_function(x, x0, t, 3)*BASE_COL;
            LED[j][i][1] = LED0[j][i][1] - wave_function(x, x0, t, 3)*BASE_COL;
            LED[j][i][2] = LED0[j][i][2] - wave_function(x, x0, t, 3)*BASE_COL;

        }
    }

}


void getrand(uint8_t color[3]){
    //generates a random color
    int lower = 0;
    int upper = 255;
    int i;
    for (i = 0; i<3; i++){
        color[i] = (rand() % (upper - lower + 1) + lower);
    }
}

void init_keys(key keys[K]){
    //maps keys to locations (only A and B for testing)
    keys[0].data = 0x1C;
    keys[0].loc = 1;
```

```
    keys[1].data = 0x32;
    keys[1].loc = 8;
}

void init_2DLED(uint8_t LED[M][N][3], uint8_t color[3]){
    //initialize LED array to be the same color
    int i;
    int j;
    for (j = 0; j< M; j++){
        for (i = 0; i < N; i++){
            LED[j][i][0] = color[0];
            LED[j][i][1] = color[1];
            LED[j][i][2] = color[2];
        }

    }

}

void sendLEDmatrix(uint8_t LED[M][N][3]){
    //send a 2D LED matrix over SPI
    int k;
    digitalWrite(GPIOB, LOAD_PIN, 1);
    for (k = 0; k< M; k++){
        sendLEDarray(LED[k]);
    }
    digitalWrite(GPIOB, LOAD_PIN, 0);
}

double wave_function(int x, int x0, float t, float c){
    //travelling Gaussian wave function
    double w = 1;
    double k = (x-c*t-x0)*(x-c*t-x0);
    double u = exp(-k/(2*w*w));
    return u;
}



int keytoloc(key keys[K], uint8_t data){
    //retrieve location for a given keypress
   int j;
   int x0;
   for (j = 0; j< K; j++){
```

```
        if (keys[j].data == data){
            x0 = keys[j].loc;
        }
    }
    return x0;
}
```

9.3 Appendix C: Verilog

```
// Final_JH_YM
// jihan23@cmc.edu jmaltsman@hmc.edu
// The FPGA receives a keypress and processes it in the ps2 module
// It then sends the bits output of the ps2 to the MCU in the spi_keyboard module
// The FPGA then receives the processed colorbits from the MCU in the spi module
// It then generates the PWM waveforms to drive LEDs in the SPI_LED_PS2 module



/////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Top-level module to receive processed colorbits from the MCU and drive the PWM waveforms to drive the LEDs
/////////////////////////////////////////////////////////////////////////////////////////////////////////////
module SPI_LED_PS2(input  logic clk,
                   input  logic sck,
                   input  logic sdi,
                   input  logic ce,
                   input  logic kclk,
                   input  logic sdi_keyboard,
                   input  logic ce_keyboard,
                   output logic sdo,
                   output logic interrupt,
                   output logic [10:0] bits,
                   output logic led,
                   output logic led_2);

   logic fclk;
   logic loadr, nextloadr;
   logic [335:0] colorbits_display;
   logic [167:0] bits_1, bits_2;
   logic ce_prev;
   logic [7:0] counter, nextcounter;

   decoder d(colorbits_display, bits_1, bits_2);
   PLL p(clk, fclk);
   spi s(sck, sdi, ce, colorbits_display);
   FPGA_ledControlUnit f_1(bits_1, loadr, fclk, led);
   FPGA_ledControlUnit f_2(bits_2, loadr, fclk, led_2);
   ps2_spi pst(clk, sck, kclk, sdi_keyboard, ce_keyboard, sdo, interrupt, bits);

   /////// assertLoad///////
   // When chip enable is just deasserted, assert the load signal for 100 clock cycles to start the FSM
   always@(posedge clk) begin
      ce_prev <= ce;
      loadr <= nextloadr;
      counter <= nextcounter;
   end

   always_comb begin
      if (ce_prev & (~ce)) begin
         nextloadr <= 1;
         nextcounter <= 0;
      end
      else if ((loadr) & (counter < 100)) begin
         nextloadr <= loadr;
         nextcounter <= counter + 1;
      end
      else if ((loadr) & (counter >= 100)) begin
         nextloadr <= 0;
         nextcounter <= 0;
      end
      else begin
         nextloadr <= 0;
         nextcounter <= 0;
      end
   end

endmodule

/////////////////////////////////////////////////////
// FSM generating the PWM waveforms for one LED strip
/////////////////////////////////////////////////////
module FPGA_ledControlUnit(input  logic [167:0] colorbits,
```

```
                          input  logic load,
                          input  logic fclk,
                          output logic led);

    logic [2:0] state, nextstate;
    logic [6:0] counter, nextcounter;
    logic [167:0] bits;
    logic [7:0] numBits, nextnumBits;

    always@(posedge fclk)
       begin
          state   <= nextstate;
          counter <= nextcounter;
          numBits <= nextnumBits;
       end

    // Next-state logic
    always_comb begin
       case(state)
          3'b000: begin
             if (load) nextstate = 3'b001;
             else nextstate = 3'b000;
          end
          3'b001: begin
             if (load) nextstate = 3'b001;
             else nextstate = 3'b010;
          end
          3'b010: begin
             if (numBits >= 168) nextstate = 3'b101;
             else if (bits[167 - numBits] == 0) nextstate = 3'b011;
             else                               nextstate = 3'b100;
          end
          3'b011: begin
             if (counter < 125) nextstate = 3'b011;
             else               nextstate = 3'b010;
          end
          3'b100: begin
             if (counter < 125) nextstate = 3'b100;
             else               nextstate = 3'b010;
          end
          3'b101: begin
             if (counter < 125) nextstate = 3'b000;
             else               nextstate = 3'b001;
          end
          default: nextstate = 3'b000;
       endcase
    end

    // PWM waveform generation
    always_comb begin
       case(state)
          3'b000: begin
             led = 0;
             nextcounter = 0;
             nextnumBits = 0;
             bits = colorbits;
          end
          3'b001: begin
             led = 0;
             nextcounter = 0;
             nextnumBits = 0;
             bits = colorbits;
          end
          3'b010: begin
             led = 0;
             nextnumBits = numBits + 1;
             nextcounter = 0;
             bits = colorbits;
          end
          3'b011: begin
             if (counter < 40) led = 1;
             else              led = 0;
             nextcounter = counter + 1;
             nextnumBits = numBits;
             bits = colorbits;
          end
```

```
            3'b100: begin
               if (counter < 80) led = 1;
               else              led = 0;
               nextcounter = counter + 1;
               nextnumBits = numBits;
               bits = colorbits;
            end
            3'b101: begin
               led = 0;
               nextcounter = counter + 1;
               nextnumBits = 0;
               bits = colorbits;
            end
            default: begin
               led = 0;
               nextcounter = 0;
               nextnumBits = 0;
               bits        = colorbits;
            end
         endcase
      end

   endmodule

   ///////////////////////////////////////////////////
   // SPI module for receiving color codes from the MCU
   ///////////////////////////////////////////////////
   module spi(input  logic sck,
              input  logic sdi,
              input  logic ce,
              output logic [335:0] colorbits);
       always_ff @(posedge sck)
           if (ce)  colorbits = {colorbits[334:0], sdi};
   endmodule


   ///////////////////////////////////////////////////////////////////////////////////////////
   // Decoder module to breakdown spi data received from the MCU into data for multiple strips
   ///////////////////////////////////////////////////////////////////////////////////////////
   module decoder(input logic [335:0] colorbits,
              output logic [167:0] bits_1,
              output logic [167:0] bits_2);

      assign bits_1 = colorbits[335:168];
      assign bits_2 = colorbits[167:0];

   endmodule

   ///////////////////////////
   // PS2 input scanning module
   ///////////////////////////
   module ps2(input  logic clk,
          input  logic kclk,
          input  logic sdi_keyboard,
          input  logic ce_keyboard,
          output logic interrupt,
          output logic [10:0] bits);

      logic fclk;
      logic [64:0] counter_ps2, nextcounter_ps2, count;
      logic [64:0] poscount, negcount;
      logic [2:0] state, nextstate;

      // configure a faster clock for more accurate sampling
      PLL p(clk, fclk);

      // Capturing the negative clock signals and sampling during those
      always@(posedge fclk) begin
         state <= nextstate;

         if(state == 3'b110) begin
            negcount <= 0;
            count <= 0;
         end
         else if(state == 3'b000) begin
            if(poscount > 3000) begin
```

```
         poscount <= 0;
         negcount <= 0;
         end
         else if(kclk) poscount <= poscount + 1;
         else if(~kclk) negcount <= negcount + 1;
      end
      else if (state == 3'b001) begin
         bits <= {bits[9:0], sdi_keyboard};
         count <= count + 1;
         poscount <= 0;
         negcount <= 0;
      end
      else if (state == 3'b010) begin
         if(negcount > 3000) begin
         poscount <= 0;
         negcount <= 0;
         end
         else if(kclk) poscount <= poscount + 1;
         else if(~kclk) negcount <= negcount + 1;
      end
      else if (state == 3'b011) begin
         poscount <= 0;
         negcount <= 0;
      end
      else if (state == 3'b100) begin
         interrupt <= 1;
         count <= 0;
      end
      else if (state == 3'b101) begin
         interrupt <= 0;
         poscount <= 0;
         negcount <= 0;
      end
      end

   // Next-state logic
   always_comb begin
      case(state)
         3'b000: if (negcount < 2000) nextstate = 3'b000;
         else nextstate = 3'b001;
         3'b001: nextstate = 3'b010;
         3'b010: begin if ((poscount >= 2000) & (count < 11)) nextstate = 3'b011;
         else if ((poscount >= 2000) & (count >= 11)) nextstate = 3'b100;
         else nextstate = 3'b010;
         end
         3'b011: nextstate = 3'b000;
         3'b100: if (ce_keyboard) nextstate = 3'b101;
         else nextstate = 3'b100;
         3'b101: nextstate = 3'b000;
         3'b110: if (poscount >= 5000) nextstate = 3'b000;
         else nextstate = 3'b110;
         default: nextstate = 3'b110;
      endcase
   end

endmodule

////////////////////////////////////////////////
// SPI to send keyboard input captured to the MCU
////////////////////////////////////////////////
module spi_keyboard(input logic sck,
                    input logic ce_keyboard,
                    input logic [7:0] colorbits,
                    output logic sdo);

    logic [3:0] counter, nextcounter;
    logic temp;
    always@(negedge sck)
      begin
         counter <= nextcounter;
      end

   always_comb
      if (ce_keyboard) begin
         nextcounter = counter + 1;
      end
```

```
      else begin
         nextcounter = 0;
      end
   assign sdo = colorbits[7-counter];

endmodule

////////////////////////////////////////////////////////////////////////////////////////////////
// Integrating PS2 input scanning and SPI to receive data from keyboard, parse it and send to the MCU
////////////////////////////////////////////////////////////////////////////////////////////////
module ps2_spi(input logic clk,
                input logic sck,
               input  logic kclk,
               input  logic sdi_keyboard,
               input  logic ce_keyboard,
               output logic sdo,
               output logic interrupt,
               output logic [10:0] bits);

   logic enable;
   logic [12:0] counter;
   logic [10:0] keybits;
   ps2 p(clk, kclk, sdi_keyboard, ce_keyboard, interrupt, keybits);
   spi_keyboard s(sck, ce_keyboard, keybits[9:2], sdo);
   assign bits = keybits;

endmodule
```