Generative Art

Owen Gillespie and Isaac Zinda, 12/12/19

Abstract

For our Microprocessor's Final Project, we set out to create a piece of art which is never the same twice. We generate a continuously evolving image using a particle simulation running on our microcontroller. The microcontroller sends this sequence of images to a field programmable gate array (FPGA) programmed with hardware we designed to control an LED array. The FPGA controls a 24x25 grid of LEDs which can each produce a quarter of a million unique colors. We hope that people will enjoy watching this art.

Introduction

Wall hangings can be boring — after all, they don't change! When a piece of art is the same day-in and day-out, it can be easily forgotten. The goal of this project was to create an LED matrix that can be hung on a wall which displays ever-evolving art. We hope that, since our art will never look the same twice, it will continue to be engaging *months* after installation.

We built our LED matrix from NeoPixel LED strips. The LEDs are mounted and placed behind a frosted piece of acrylic so their light blends together. The matrix is 24x25 LEDs and its dimensions are .4 meters by .4 meters. We use the microcontroller to generate the image data and send this to the FPGA, which writes to the LEDs in parallel and notifies the microcontroller when the frame has been displayed and the next one can be sent.

One of our goals was to design hardware capable of controlling may more LEDs than we could afford to purchase. Our FPGA can control up to 24 strips of LEDs in parallel, allowing us to drive more than 10k LEDs at 60 FPS. That would allow us to drive a four square meter LED matrix, assuming that the microcontroller RAM is not a limiting factor. If we tried to use the microcontroller alone to drive even close to this many LEDs, it would use all of the CPU cycles writing the signal, and would have no time to receive or generate the output data.

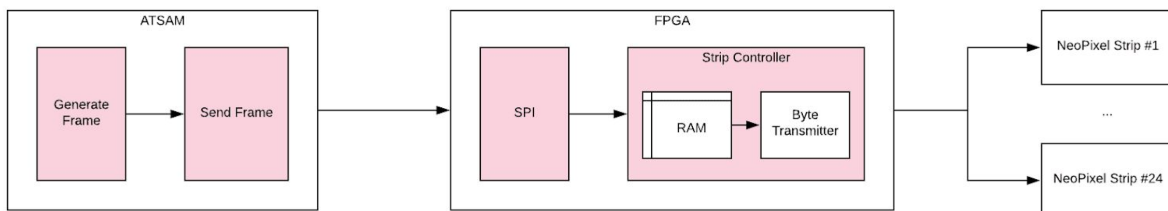

Figure 1: Block diagram overview of our entire system

New Hardware

This project revolves around strips of NeoPixels (WS2812B LEDs), which are different from any hardware we have used before. A single NeoPixel is composed of red, green, and blue LEDs along with a surface mounted chip that drives them. We will refer to one of these units as a pixel, and a collection of them wired together as a strip. NeoPixels use a self-clocking data protocol so that an entire strip can be controlled using only 1 pin. In particular, there are three signals that can be sent: A bit of data (either a 1 or 0) or a reset. Bits are sent by holding the signal high for a specified time, and then low for a specified time. Depending on how long the signal stays high or low, it is interpreted as a 0 or a 1 by the Neopixels. This pattern is shown below in Figure 2 with the exact timings in Table 1.

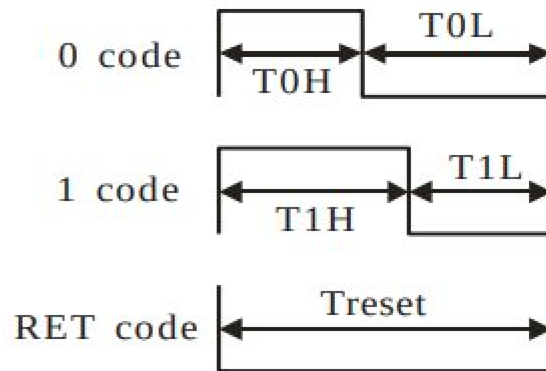

Figure 2: Waveforms for NeoPixel data protocol

T0H	0 code ,high voltage time	0.4us	±150ns
T1H	1 code ,high voltage time	0.8us	±150ns
T0L	0 code , low voltage time	0.85us	±150ns
T1L	1 code ,low voltage time	0.45us	±150ns
RES	low voltage time	Above 50μs	

Table 1: Timing specifications for NeoPixel data protocol

In order to set the strip to a sequence of colors, the user transmits the desired color of each pixel in the strip in order by sending 24 bits of data (1 byte each for red, green, and blue) for any number of pixels consecutively, and then sends the reset code to indicate that all of the data has been sent out and the Neopixels should update to the colors sent and prepare for a new data transmission. Because each pixel has a chip which latches onto the first 24 bits of the transmission, refreshes the rest of the signal, and forwards it to the next pixel, a single pin can in theory control any number of pixels. Figure 3 shows an example of how the data propagates down a strip below.

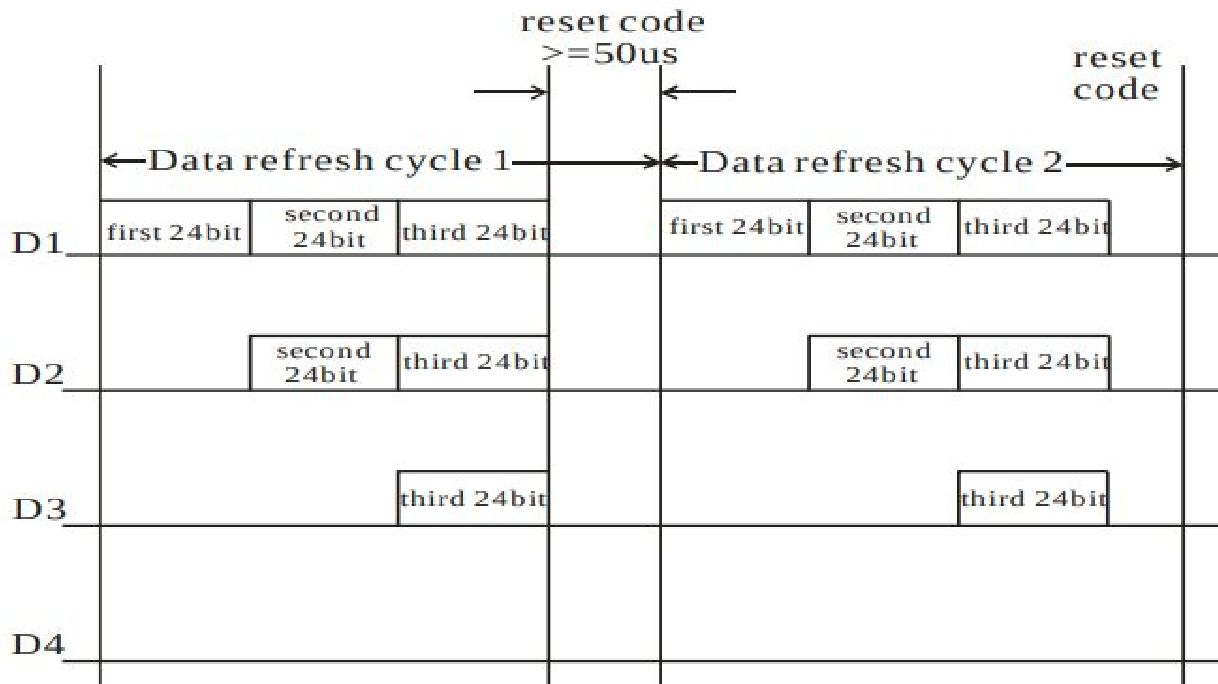

Figure 3: NeoPixel strip example data transmission

However, the limiting factor in chaining NeoPixels becomes the refresh rate. Because the self-clocking data protocol only allows for the transmission of 1 bit per 1.35 μs, the amount of time it takes to send a "frame" of data (the color data for each pixel in a strip) scales linearly with the number of pixels. This effectively limits a microcontroller to controlling ~1000 pixels at 30 FPS. By controlling many strips of NeoPixels in parallel with an FPGA, we can run more than 20 times as many pixels before this constraint becomes a problem.

Microcontroller Design

Art Algorithm Explanation

Art is generated on the microcontroller using a particle simulation. Particles are placed randomly on the screen with a random velocity and a random color. There are twenty particles at all times — if a particle ever leaves the screen, it is replaced with a new particle, randomized in the fashion described above. Particles are affected by a "gravity" acceleration, so they tend to fall down.

Every frame, each pixel on the screen is set the color of the nearest particle. The microcontroller then waits until the flushing signal from the FPGA is low, indicating that the FPGA is ready to accept new pixel data. At this point, calculated pixel values are sent to the FPGA over a custom serial protocol, discussed in the "FPGA Design" section. For a more detailed description of the microcontroller's functioning, see Figure 5 below.

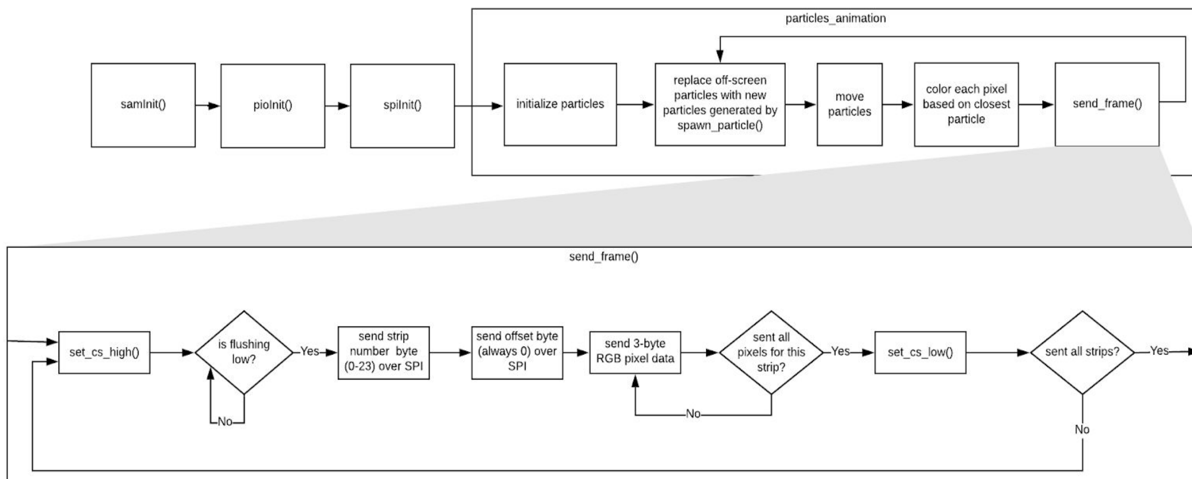

Figure 5: Flow chart of the algorithm we used to generate art and send it to the FPGA

Refresh Rate

In this section, we calculate how long it takes the microcontroller to generate art and how long it takes to send a frame over SPI. First, we calculate the number of bits which need to be sent to update the screen:

$$24\ (strips)\ \times 25\ (LED\ /\ strip)\ \times 24\ (bits\ /\ LED)\ =\ 14400\ bits$$

We have configured SPI to use a clock divider of 36. Since the clock runs at 40 MHz, the SPI clock is ~1 MHz. We can combine this information with the previous equation to find the time taken to send each frame:

$$14400\ (bits\ /\ frame)\ *\ 1/1000000\ (second\ /\ bits)\ =\ 14.4ms$$

Using a scope we have measured that our refresh period is 60ms, so it must takes around $45ms$ to generate our art. If we wish to achieve a faster refresh rate in the future, we should primarily investigate ways to speed up the particle simulation in software, as well as speeding up SPI.

Testing Image Generation

We wanted to be able to develop our art quickly, without needing to flash our microcontroller or be in the lab at all. In order to do this, we built a software simulation that pretends to be our FPGA / NeoPixel hardware but is actually entirely software-based.

Ordinarily, the method `send_frame` is used to send a 24x25 frame to the FPGA to be displayed on the NeoPixels. At the top of our program, we `#define` a constant called

TESTING. If we have #define TESTING 0, we are not testing and the real send_frame is used which actually send a frame to our hardware over SPI. If we have written #define TESTING 1, we are testing and instead of sending the frame to actual hardware, send_frame outputs the frame to files on disk. These files on disk are then read by a Python program, which draws the frame to a computer window. Figure 6, included below, is a screen capture from this simulator:

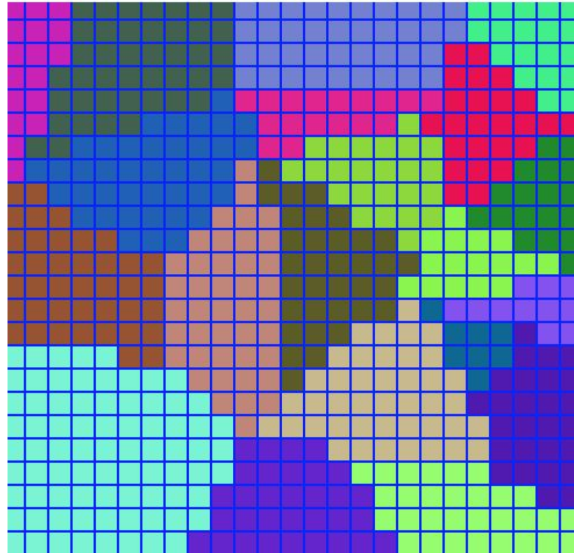

Figure 6: A screen capture of our software LED simulator, running the particles animation

FPGA Design

Our motivating reason for using the FPGA in this project was to create a device which could drive thousands of NeoPixels at a reasonable frame rate. Since a single strip is limited to a length of about 1000 pixels at 30 FPS, our FPGA design is fundamentally about allowing many strips to be driven in parallel.

At a high level, the FPGA design for this process is composed of an SPI module which receives commands from the microcontroller, and a series of 24 strip controllers which maintain the state of one strip and handle writing data to it. The SPI controller reads in commands from the microcontroller over SPI. The high level modules and control signals are shown in Figure 7.

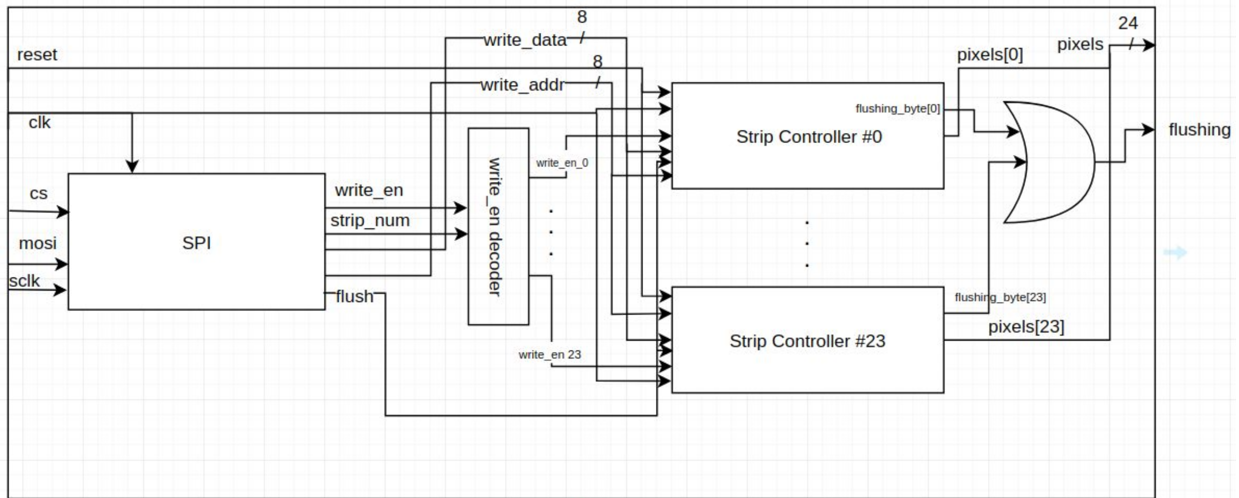

Figure 7. FPGA design high level modules and control signals

The command protocol consists of a write command which specifies a strip number and a pixel offset before sending color data for any number of pixels as shown in Figure 8. The command is terminated when chip select goes low. This will be interpreted as writing to the pixels on the strip starting at the given offset and increasing the index into the strip by one every 3 bytes of color data. As the SPI module reads this command, it sets `write_en` to high, `strip_num` to the appropriate strip number, and changes `write_data` and `write_addr` to reflect the appropriate data for the strip controller to write to it's internal RAM and the correct offset for that data.

Strip Number	**Offset**	**Red**	**Green**	**Blue**
1 byte, 0-23	1 byte, 0-255	1 byte, 0-255	1 byte, 0-255	1 byte, 0-255

Figure 8. Write command data format

When the SPI controller receives the flush command (a single magic byte as shown in Figure 9) it sets the flush signal high for one cycle.

Figure 9. Flush command data format

Between the SPI module and the strip controllers, there is some logic that does a one-hot encoding of the strip number when `write_en` is high. In the future this could be incorporated into the SPI module.

Each strip controller maintains the state of the strip in a RAM module. When it’s `write_en` signal goes high, it writes `write_data` in RAM to `write_addr`. When flush goes high, it sets `byte_flushing` high, outputs the entire contents of the strip in the WS2812B data encoding over it’s pixels output, holds the pixels output low for 50 μs to encode a reset instruction, and then sets its `byte_flushing` low again.

Schematic

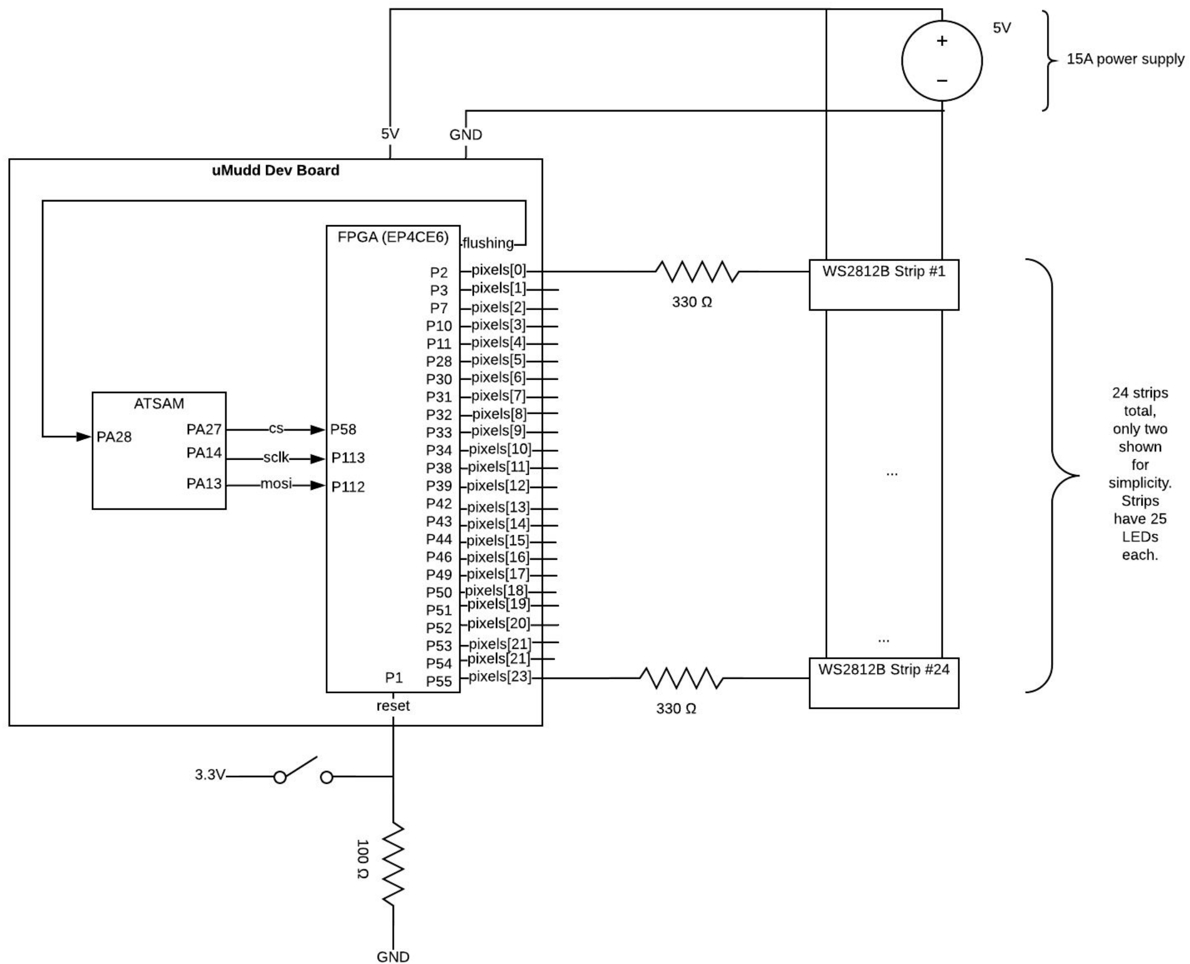

Figure 4: Schematic of the circuitry we build

Results

All aspects of this project worked out to specification. We have fabricated a 24x25 LED matrix and mounted this behind a piece of frosted plastic to diffuse the light. We generate art on the microcontroller and then send it to the FPGA, which writes to 24 LED strips in parallel.

We improved upon our initial proposal for transferring data between the microcontroller and the FPGA. We added a CS pin in addition to the originally proposed data and clock pins in order to support burst mode.

The refresh rate we achieved in our finished product is about 17Hz. As explained in our "Refresh Rate" subsection, this low refresh rate is primarily due to the time taken to generate a complex pattern. However, the FPGA design succeeded in writing to 600 LEDs in about 1 ms, which would have allowed a 400-500 Hz refresh rate or been able to drive thousands of LEDs at an acceptable refresh rate if it were driven by a faster microcontroller over a faster connection.

Future work could involve modifying the generated pattern based on user input. We propose adding additional visualizations, as well as reacting to viewer input such as by allowing the viewer to change the direction of gravity, or modifying the color of newly spawned particles based on ambient noise.

References

WS2812B Datasheet — https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf
ATSAM4S Family Datasheet —
http://pages.hmc.edu/harris/class/e155/ATSAM4S_Family_Datasheet.pdf
μMudd Board Schematic —
http://pages.hmc.edu/harris/class/e155/uMuddMarkV_1Logic_print.pdf

BOM

When a price is not listed, the items were taken free of charge from the μPs lab, makerspace, or engineering stockroom.

Mechanical

- 1/16" thick 18" x 24" acrylic sheet — $11.48
- ¾" thick 16" x 24" wood sheet — $7
- (4) 1" nylon spacers — $3
- Frosted privacy film — $9.99
- (4) 2" 6/32 bolts — $3
- (4) 6/32 nuts
- (4) metal brackets
- (8) wood screws

Electrical

- μMudd Development Board (and supplied breadboard)
- (2) WS2812B 300 LED Strips — $61.76
- 15A 5V Power Supply (includes female adapter) — $25.99
- (12) wire caps
- ~3 meters 22 gauge single-stranded wire

- ~1 meter 16 gauge wire
- 100Ω resistor
- (24) 330Ω resistor
- Pushbutton
- 1000 μF capacitor

Appendix

So that code could be properly formatted using the correct font and syntax highlighting, the SystemVerilog and C code has been attached to the end of this document.

```
1 /*
2 main.c
3 */
4
5 // outputs frame to local files when we are testing
6 // outputs frame to strips over SPI when we aren't testing
7 #define TESTING 0
8
9 #include "particles.h"
10
11 // need to redefine this sometimes because ATSAM's version of math.h doesn't
12 // include this constant
13 #ifndef M_PI
14 #define M_PI 3.1415
15 #endif
16
17 // to simplify boolean logic
18 #define TRUE 1
19 #define FALSE 0
20
21 typedef unsigned char byte;
22
23
24 int main() {
25   #if TESTING == 0
26     samInit();
27     pioInit();
28     spiInit(72, 0, 1);
29     pioPinMode(CS_PIN, PIO_OUTPUT);
30     pioPinMode(FLUSHING_PIN, PIO_INPUT);
31   #endif
32
33     particles_animation();
34 }
35
```

```c
/*
particles.h

Because we want to work with integers and not floats, all velocities and
posititions are measured in 1/1024th pixel units.
*/

#include "screen.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NUM_PARTICLES 20

// units per pixel
#define UPS 1024
// max velocity (in units / frame)
// note that this is not pixels / frame
#define MAX_VEL_UNITS 50

// how much velocity changes by each frame in y direction
#define GRAVITY 10
#define FALSE 0
#define TRUE 1

typedef struct {
    int x_pos;
    int y_pos;
    int x_vel;
    int y_vel;
    color col;
    int active;
} particle;


particle particles[NUM_PARTICLES];

int randrange(int min, int max) {
    return (rand() % (max-min)) + min;
}
particle spawn_particle() {
    int rand_x_vel = randrange(-MAX_VEL_UNITS, MAX_VEL_UNITS);
    int rand_y_vel = randrange(-MAX_VEL_UNITS, MAX_VEL_UNITS);

    int rand_x_pos = randrange(0, (WIDTH * UPS));
    int rand_y_pos = randrange(0, (WIDTH * UPS));

    color rand_color = (color){(byte) (rand() % 256), (byte) (rand() % 256),
(byte) (rand() % 256), 255};

    return (particle){rand_x_pos, rand_y_pos, // position
        rand_x_vel, rand_y_vel,
        rand_color, // color
        TRUE}; // active?
}

particle get_closest_particle(int x_pos, int y_pos) {
    particle closest_particle;
    // larger than max distance
    unsigned int closest_particle_distance = 4294967295; // max value of int
```

```
60
61     for (int i = 0; i < NUM_PARTICLES; i++) {
62         int distance = (x_pos - particles[i].x_pos) * (x_pos -
particles[i].x_pos) +
63                        (y_pos - particles[i].y_pos) * (y_pos -
particles[i].y_pos);
64
65         // this will make the pattern look worse, but will save LOTS of
cycles
66         // int distance = abs(x_pos - particles[i].x_pos) + abs(y_pos -
particles[i].y_pos);
67
68         if (distance < closest_particle_distance) {
69             closest_particle_distance = distance;
70             closest_particle = particles[i];
71         }
72     }
73
74     return closest_particle;
75 }
76
77 float p;
78
79 void particles_animation() {
80     // initialize all particles to
81     for (int s = 0; s < NUM_PARTICLES; s++) {
82         particles[s].active = FALSE;
83     }
84
85     while (TRUE) {
86         // for (int i = 0; i < 10000; i++) { // 100000 2 fps
87         //  p = sqrt(i);
88         //}
89
90         for (int s = 0; s < NUM_PARTICLES; s++) {
91             // add this particle if it has fallen off of the screen
92             if (particles[s].active == FALSE) {
93                 // spawn a new particle
94                 particles[s] = spawn_particle();
95             }
96
97             // move particles
98             particles[s].x_pos += particles[s].x_vel;
99             particles[s].y_pos += particles[s].y_vel;
100
101            particles[s].y_vel += GRAVITY;
102        }
103
104        // color pixels based on which particle is closest
105        for (int x = 0; x < WIDTH; x++) {
106            for (int y = 0; y < HEIGHT; y++) {
107                color pixel_color = get_closest_particle(x*UPS, y*UPS).col;
108                set_screen_color(y, x, pixel_color);
109            }
110        }
111
112        send_frame();
113
114        // remove particles which have fallen off of the screen
115        for (int s = 0; s < NUM_PARTICLES; s++) {
```

```
116             if (particles[s].x_pos >= WIDTH * UPS || particles[s].x_pos < 0
||
117                 particles[s].y_pos >= HEIGHT * UPS || particles[s].y_pos < 0)
118             {
119                 particles[s].active = FALSE;
120
121             }
122         }
123     }
124 }
125
```

```c
/*
screen.h
*/

#include <stdio.h>

typedef unsigned char byte;
typedef struct {
  byte red;
  byte green;
  byte blue;
  byte alpha;
} color;

// number of cols
#define WIDTH 25

// number of rows
#define HEIGHT 24

// we send 3 pixels -- red, green, blue -- over SPI
#define PIXEL_SIZE 3

// constant we send when we want to flush
#define FLASH_CONSTANT 70 // 0b10101010

// all colors are bitshifted by this constant, which acts like a dimmer
#define BRIGHTNESS_SHIFT 2

#define RED 0
#define GREEN 1
#define BLUE 2

#if TESTING == 1
#include <sys/stat.h>
#include <sys/types.h>
#endif

#if TESTING == 0
#include "SAM4S4B_libs/SAM4S4B.h"
#endif

#define FLUSHING_PIN PIO_PA26
#define CS_PIN PIO_PA27

byte screen[HEIGHT * WIDTH * PIXEL_SIZE];

void set_screen_color(unsigned char h, unsigned char w, color pixel_color){
  screen[(h * WIDTH * PIXEL_SIZE) + (w * PIXEL_SIZE)] = pixel_color.red;
  screen[(h * WIDTH * PIXEL_SIZE) + (w * PIXEL_SIZE) + 1] = pixel_color.green;
  screen[(h * WIDTH * PIXEL_SIZE) + (w * PIXEL_SIZE) + 2] = pixel_color.blue;
}

void set_screen(unsigned char h, unsigned char w, unsigned char red, unsigned char green, unsigned char blue){
  screen[(h * WIDTH * PIXEL_SIZE) + (w * PIXEL_SIZE)] = red;
  screen[(h * WIDTH * PIXEL_SIZE) + (w * PIXEL_SIZE) + 1] = green;
  screen[(h * WIDTH * PIXEL_SIZE) + (w * PIXEL_SIZE) + 2] = blue;
}
```

```
59
60 byte get_screen_red(unsigned char h, unsigned char w){
61   return screen[(h * WIDTH * PIXEL_SIZE) + (w * PIXEL_SIZE)];
62 }
63
64 byte get_screen_green(unsigned char h, unsigned char w){
65   return screen[(h * WIDTH * PIXEL_SIZE) + (w * PIXEL_SIZE) + 1];
66 }
67
68 byte get_screen_blue(unsigned char h, unsigned char w){
69   return screen[(h * WIDTH * PIXEL_SIZE) + (w * PIXEL_SIZE) + 2];
70 }
71
72 /*
73 Code which is used in order to send frames over SPI (not for testing).
74 */
75 #if TESTING == 0
76 void spi_send_byte(byte data){
77   while (pioDigitalRead(FLUSHING_PIN)) {}
78   spiSendReceive(data);
79 }
80
81 void set_cs_high(){
82   pioDigitalWrite(CS_PIN, PIO_HIGH);
83 }
84
85 void set_cs_low(){
86   pioDigitalWrite(CS_PIN, PIO_LOW);
87 }
88
89 void flush(){
90   set_cs_high();
91   spi_send_byte(FLASH_CONSTANT);
92   set_cs_low();
93 }
94
95 void send_strip(byte strip_number, byte strip_data[WIDTH][PIXEL_SIZE]){
96   set_cs_high();
97   spi_send_byte(strip_number); // Start by sending strip number
98   spi_send_byte(0); // 0 offset
99   for (int w = 0; w < WIDTH; w++) {
100     for (int p = 0; p < PIXEL_SIZE; p++){
101       spi_send_byte((char) strip_data[w][p]); // send all of the data from
    the strip using "burst mode"
102     }
103   }
104   set_cs_low();
105 }
106 #endif
107
108 /*
109 Code which is used to save frames to files (for testing).
110 */
111 #if TESTING == 0
112 void send_frame(){
113   for (int h = 0; h < HEIGHT; h++) {
114       set_cs_high();
115       spi_send_byte(h); // Start by sending strip number
116       spi_send_byte(0); // 0 offset
117       for (int w = 0; w < WIDTH; w++) {
```

```
118         for (int p = 0; p < PIXEL_SIZE; p++){
119           spi_send_byte(screen[(h * WIDTH * PIXEL_SIZE) + (w * PIXEL_SIZE) +
    p] << BRIGHTNESS_SHIFT); // send all of the data from the strip using "burst
    mode"
120         }
121       }
122       set_cs_low();
123   }
124   flush();
125 }
126 #endif
127
128 #if TESTING == 1
129 void send_frame() {
130   static int frame_number = 0;
131   char frame_name[50];
132
133   sprintf(frame_name, "frames/frame_%d.txt", frame_number);
134
135   int result = mkdir("frames", 0777); // create frames dir if it doesn't
    exist
136   FILE* fd = fopen(frame_name, "w+");
137
138   for (int h = 0; h < HEIGHT; h++) {
139     for (int w = 0; w < WIDTH; w++) {
140       fprintf(fd, "%u,%u,%u\n", get_screen_red(h, w), get_screen_green(h, w),
    get_screen_blue(h, w));
141     }
142   }
143
144   fclose(fd);
145   frame_number += 1;
146 }
147 #endif
148
```

```
1  module top (input logic clk, reset, sclk, mosi, cs,
2             output logic [23:0] pixels,
3             output logic flushing, cs_out,
4             output logic [1:0] spi_state);
5
6    assign cs_out = cs;
7
8    neopixel_driver #(75) neopixel_driver1(clk, reset, sclk, mosi, cs, pixels,
   flushing, spi_state[0], spi_state[1]);
9  endmodule
10
11 module neopixel_driver #(parameter STRIP_LENGTH)
12            (input logic clk, reset, sclk, mosi, cs,
13             output logic [23:0] pixels,
14             output logic flushing);
15
16   logic flush, write_en;
17   logic [7:0] write_data, write_addr;
18   logic [4:0] strip_num;
19
20   spi spi(clk, sclk, mosi, cs, flush, write_en, write_data, write_addr,
   strip_num);
21
22   logic [23:0] byte_flushing, write_en_list;
23   assign flushing = & byte_flushing;
24
25   assign write_en_list[0] = (strip_num == 5'd0) ? write_en : 1'b0;
26   assign write_en_list[1] = (strip_num == 5'd1) ? write_en : 1'b0;
27   assign write_en_list[2] = (strip_num == 5'd2) ? write_en : 1'b0;
28   assign write_en_list[3] = (strip_num == 5'd3) ? write_en : 1'b0;
29   assign write_en_list[4] = (strip_num == 5'd4) ? write_en : 1'b0;
30   assign write_en_list[5] = (strip_num == 5'd5) ? write_en : 1'b0;
31   assign write_en_list[6] = (strip_num == 5'd6) ? write_en : 1'b0;
32   assign write_en_list[7] = (strip_num == 5'd7) ? write_en : 1'b0;
33   assign write_en_list[8] = (strip_num == 5'd8) ? write_en : 1'b0;
34   assign write_en_list[9] = (strip_num == 5'd9) ? write_en : 1'b0;
35   assign write_en_list[10] = (strip_num == 5'd10) ? write_en : 1'b0;
36   assign write_en_list[11] = (strip_num == 5'd11) ? write_en : 1'b0;
37   assign write_en_list[12] = (strip_num == 5'd12) ? write_en : 1'b0;
38   assign write_en_list[13] = (strip_num == 5'd13) ? write_en : 1'b0;
39   assign write_en_list[14] = (strip_num == 5'd14) ? write_en : 1'b0;
40   assign write_en_list[15] = (strip_num == 5'd15) ? write_en : 1'b0;
41   assign write_en_list[16] = (strip_num == 5'd16) ? write_en : 1'b0;
42   assign write_en_list[17] = (strip_num == 5'd17) ? write_en : 1'b0;
43   assign write_en_list[18] = (strip_num == 5'd18) ? write_en : 1'b0;
44   assign write_en_list[19] = (strip_num == 5'd19) ? write_en : 1'b0;
45   assign write_en_list[20] = (strip_num == 5'd20) ? write_en : 1'b0;
46   assign write_en_list[21] = (strip_num == 5'd21) ? write_en : 1'b0;
47   assign write_en_list[22] = (strip_num == 5'd22) ? write_en : 1'b0;
48   assign write_en_list[23] = (strip_num == 5'd23) ? write_en : 1'b0;
49
50   strip_controller #(STRIP_LENGTH) strip_controller0(clk, reset,
   write_en_list[0], flush, write_data, write_addr, byte_flushing[0],
   pixels[0]);
51   strip_controller #(STRIP_LENGTH) strip_controller1(clk, reset,
   write_en_list[1], flush, write_data, write_addr, byte_flushing[1],
   pixels[1]);
52   strip_controller #(STRIP_LENGTH) strip_controller2(clk, reset,
   write_en_list[2], flush, write_data, write_addr, byte_flushing[2],
   pixels[2]);
```

```
53   strip_controller #(STRIP_LENGTH) strip_controller3(clk, reset,
write_en_list[3], flush, write_data, write_addr, byte_flushing[3],
pixels[3]);
54   strip_controller #(STRIP_LENGTH) strip_controller4(clk, reset,
write_en_list[4], flush, write_data, write_addr, byte_flushing[4],
pixels[4]);
55   strip_controller #(STRIP_LENGTH) strip_controller5(clk, reset,
write_en_list[5], flush, write_data, write_addr, byte_flushing[5],
pixels[5]);
56   strip_controller #(STRIP_LENGTH) strip_controller6(clk, reset,
write_en_list[6], flush, write_data, write_addr, byte_flushing[6],
pixels[6]);
57   strip_controller #(STRIP_LENGTH) strip_controller7(clk, reset,
write_en_list[7], flush, write_data, write_addr, byte_flushing[7],
pixels[7]);
58   strip_controller #(STRIP_LENGTH) strip_controller8(clk, reset,
write_en_list[8], flush, write_data, write_addr, byte_flushing[8],
pixels[8]);
59   strip_controller #(STRIP_LENGTH) strip_controller9(clk, reset,
write_en_list[9], flush, write_data, write_addr, byte_flushing[9],
pixels[9]);
60   strip_controller #(STRIP_LENGTH) strip_controller10(clk, reset,
write_en_list[10], flush, write_data, write_addr, byte_flushing[10],
pixels[10]);
61   strip_controller #(STRIP_LENGTH) strip_controller11(clk, reset,
write_en_list[11], flush, write_data, write_addr, byte_flushing[11],
pixels[11]);
62   strip_controller #(STRIP_LENGTH) strip_controller12(clk, reset,
write_en_list[12], flush, write_data, write_addr, byte_flushing[12],
pixels[12]);
63   strip_controller #(STRIP_LENGTH) strip_controller13(clk, reset,
write_en_list[13], flush, write_data, write_addr, byte_flushing[13],
pixels[13]);
64   strip_controller #(STRIP_LENGTH) strip_controller14(clk, reset,
write_en_list[14], flush, write_data, write_addr, byte_flushing[14],
pixels[14]);
65   strip_controller #(STRIP_LENGTH) strip_controller15(clk, reset,
write_en_list[15], flush, write_data, write_addr, byte_flushing[15],
pixels[15]);
66   strip_controller #(STRIP_LENGTH) strip_controller16(clk, reset,
write_en_list[16], flush, write_data, write_addr, byte_flushing[16],
pixels[16]);
67   strip_controller #(STRIP_LENGTH) strip_controller17(clk, reset,
write_en_list[17], flush, write_data, write_addr, byte_flushing[17],
pixels[17]);
68   strip_controller #(STRIP_LENGTH) strip_controller18(clk, reset,
write_en_list[18], flush, write_data, write_addr, byte_flushing[18],
pixels[18]);
69   strip_controller #(STRIP_LENGTH) strip_controller19(clk, reset,
write_en_list[19], flush, write_data, write_addr, byte_flushing[19],
pixels[19]);
70   strip_controller #(STRIP_LENGTH) strip_controller20(clk, reset,
write_en_list[20], flush, write_data, write_addr, byte_flushing[20],
pixels[20]);
71   strip_controller #(STRIP_LENGTH) strip_controller21(clk, reset,
write_en_list[21], flush, write_data, write_addr, byte_flushing[21],
pixels[21]);
72   strip_controller #(STRIP_LENGTH) strip_controller22(clk, reset,
write_en_list[22], flush, write_data, write_addr, byte_flushing[22],
pixels[22]);
```

```
73   strip_controller #(STRIP_LENGTH) strip_controller23(clk, reset,
write_en_list[23], flush, write_data, write_addr, byte_flushing[23],
pixels[23]);
74
75
76
77 endmodule
78
79 module RAM(input logic clk,
80            input logic [7:0] write_addr, write_data,
81         input logic write_en,
82         input logic [7:0] read_addr,
83         output logic [7:0] read_data);
84
85   logic [7:0] mem[255:0];
86
87   always_ff @(posedge clk) begin
88       read_data <= mem[read_addr];
89       if (write_en) mem[write_addr] <= write_data;
90     end
91 endmodule
92
93
94 module bit_transmitter (input logic clk, reset, start, in,
95                 output logic out, done);
96
97   typedef enum logic [1:0] {WAITING, WRITING} statetype;
98   statetype state, next_state;
99
100   // longest it needs to be is 1.25 uS == 50 cycles
101   // low_level is the number of cycles we hold low
102   logic [5:0] counter, next_counter, low_level;
103
104   // 18 cycles == .45 uS
105   // 34 cycles == .85 uS
106   assign low_level = in ? 6'd18 : 6'd34;
107
108   // write 0 when there's no valid output
109   assign out = (state == WRITING) ? (counter > low_level) : 1'b0;
110   assign done = state == WAITING;
111
112   always_ff @(posedge clk, posedge reset)
113     if (reset) begin
114         state <= WAITING;
115
116       end
117     else
118       begin
119         state <= next_state;
120         counter <= next_counter;
121       end
122
123   always_comb
124     case (state)
125       WAITING:
126         if (start) begin
127           next_state = WRITING;
128           next_counter = 6'd50;
129         end else begin
130           next_state = WAITING;
```

```
131           next_counter = 6'dx; // doesn't matter
132         end
133       WRITING:
134         if (counter == 0) begin
135           next_state = WAITING;
136           next_counter = 6'dx;
137         end else begin
138           next_state = WRITING;
139           next_counter = counter - 6'd1;
140         end
141     endcase
142 endmodule
143 
144 
145 // when start goes high, we transmit
146 module byte_transmitter (input logic clk, reset, start,
147                   input logic [7:0] in,
148                  output logic out, done);
149 
150   typedef enum logic [1:0] {WAITING, WRITING_START, WRITING_BLOCK} statetype;
151   statetype state, next_state;
152 
153   logic [2:0] counter, next_counter;
154 
155   logic bit_start, bit_in, bit_done;
156 
157   bit_transmitter bit_trasmitter1(clk, reset, bit_start, bit_in, out,
    bit_done);
158 
159   logic store_byte;
160   logic [7:0] byte_read;
161 
162 
163   always_ff @(posedge clk)
164     if (store_byte) byte_read <= in;
165 
166 
167   always_ff @(posedge clk, posedge reset)
168     if (reset) begin
169         state <= WAITING;
170       end
171     else
172       begin
173         state <= next_state;
174         counter <= next_counter;
175       end
176 
177   assign done = state == WAITING;
178 
179   always_comb
180     case(state)
181       WAITING:
182         begin
183           bit_in = 1'bx;
184           bit_start= 1'b0;
185 
186           if(start) begin
187               store_byte = 1'b1;
188               next_counter = 3'b000;
189               next_state = WRITING_START;
```

```
190
191             end
192           else begin
193               store_byte = 1'b0;
194               next_counter = 3'bxxx;
195               next_state = WAITING;
196             end
197         end
198
199       WRITING_START:
200         begin
201           store_byte = 1'b0;
202           next_state = WRITING_BLOCK;
203           next_counter = counter;
204           bit_in = byte_read[counter];
205           bit_start = 1'b1;
206         end
207
208       WRITING_BLOCK:
209         // if the bit controller finished writing its bit
210         if (bit_done) begin
211             if (counter == 3'b111) begin
212                 store_byte = 1'b0;
213                 next_state = WAITING;
214                 next_counter = 3'bxxx;
215                 bit_in = 1'bx;
216                 bit_start = 1'b0;
217               end
218             else begin
219                 store_byte = 1'b0;
220                 next_state = WRITING_START;
221                 next_counter = counter + 3'b001;
222                 bit_in = 1'bx; // no need to set this valid here
223                 bit_start = 1'b1;
224               end
225           end
226         // if the bit controller is still writing its bit
227         else begin
228             store_byte = 1'b0;
229             next_state = WRITING_BLOCK;
230             next_counter = counter;
231             bit_in = byte_read[counter];
232             bit_start = 1'b0;
233           end
234     endcase
235 endmodule
236
237
238 module strip_controller #(parameter STRIP_LENGTH)
239                          (input logic clk, reset, write_en, flush,
240                           input logic [7:0] write_data, write_addr,
241                   output logic flushing, data_out);
242
243   logic [7:0] read_addr, next_read_addr, read_data;
244
245   RAM strip_data(clk, write_addr, write_data, write_en, next_read_addr,
    read_data);
246
247   logic byte_start, byte_out, byte_done, next_byte_start;
248   // Counts to 2000 cycles = 50 microseconds
```

```
249   logic [10:0] waiting_counter, next_waiting_counter;
250
251
252   byte_transmitter byte_transmitter1(clk, reset, byte_start, read_data,
byte_out, byte_done);
253
254   // when flush goes high, enter "flushing" state
255   typedef enum logic [1:0] {LOADING, FLUSHING_START, FLUSHING_BLOCK, WAITING}
statetype;
256   statetype state, next_state;
257
258   always_ff @(posedge clk, posedge reset)
259     if (reset) begin
260         state <= LOADING;
261         read_addr <= 8'hxx;
262         byte_start <= 1'bx;
263         waiting_counter <= 11'bx;
264       end
265     else
266       begin
267         state <= next_state;
268         read_addr <= next_read_addr;
269         byte_start <= next_byte_start;
270         waiting_counter <= next_waiting_counter;
271       end
272
273   assign flushing = (state == FLUSHING_START || state == FLUSHING_BLOCK ||
state == WAITING);
274   assign data_out = byte_out;
275
276   always_comb
277     case(state)
278       LOADING:
279         if(flush) begin
280             next_state = FLUSHING_START;
281             next_read_addr = 8'h00;
282             next_byte_start = 1'b1;
283             next_waiting_counter = 11'bx;
284           end
285         else begin
286             next_state = LOADING;
287             next_read_addr = 8'hxx;
288             next_byte_start = 1'b0;
289             next_waiting_counter = 11'bx;
290           end
291       FLUSHING_START:
292         begin
293           next_state = FLUSHING_BLOCK;
294           next_read_addr = read_addr;
295           next_byte_start = 1'b0;
296           next_waiting_counter = 11'bx;
297         end
298       FLUSHING_BLOCK:
299         if(byte_done) begin
300             if (read_addr == (STRIP_LENGTH - 1)) begin
301                 next_state = WAITING;
302                 next_read_addr = 8'hxx;
303                 next_byte_start = 1'b0;
304                 next_waiting_counter = 11'b0;
305               end
```

```
306              else begin
307                next_state = FLUSHING_START;
308                next_read_addr = read_addr + 8'h01;;
309                next_byte_start = 1'b1;
310                next_waiting_counter = 11'bx;
311                end
312            end
313          else begin
314              next_state = FLUSHING_BLOCK;
315              next_read_addr = read_addr;
316              next_byte_start = 1'b0;
317              next_waiting_counter = 11'bx;
318            end
319        WAITING:
320          if (waiting_counter == 11'd2000) begin
321              next_state = LOADING;
322              next_read_addr = 8'hxx;
323              next_byte_start = 1'b0;
324              next_waiting_counter = 11'bx;
325            end
326          else begin
327              next_state = WAITING;
328              next_read_addr = 8'hxx;
329              next_byte_start = 1'b0;
330              next_waiting_counter = waiting_counter + 11'b1;
331            end
332        default: begin
333            next_state = state;
334            next_read_addr = 8'hxx;
335            next_byte_start = 1'bx;
336            next_waiting_counter = 11'bx;
337          end
338      endcase
339 endmodule
340
341
342
343 module spi(input logic clk, sclk, mosi, cs,
344         output logic flush, write_en,
345         output logic [7:0] write_data,
346         output logic [7:0] write_offset,
347         output logic [4:0] strip_num);
348
349   logic [7:0] byte_read;
350   logic strip_num_en, offset_en, offset_incr, byte_done, gated_cs;
351
352
353   // this outputs everything in the clk clock domain
354   // it outputs byte_read, which is the byte read, and raises
355   // new_byte for 1 cycle after byte_read becomes valid
356
357   // TODO: gated cs is very hakcy, fix this ;)
358   spi_byte_reader spi_byte_reader1(clk, sclk, cs, mosi, byte_done, gated_cs,
byte_read);
359
360   assign write_data = byte_read;
361
362   spi_controller spi_controller1(clk, gated_cs, byte_done, byte_read,
strip_num_en, offset_en, write_en, offset_incr, flush);
363
```

```
364
365   always_ff @(posedge clk)
366     if (strip_num_en) strip_num <= byte_read[4:0];
367
368   // offset holds the offset from the address pointer
369   // where we will write data
370   always_ff @(posedge clk)
371     if (offset_en) begin
372         if (offset_incr) write_offset <= write_offset + 8'b1;
373         else write_offset <= byte_read;
374       end
375 endmodule
376
377 // all outputs are synchronized to clk
378 module spi_byte_reader(input logic clk,
379                 input logic sclk, cs, mosi,
380                 output logic byte_done, gated_cs,
381                 output logic [7:0] byte_read);
382
383   logic [7:0] sclk_byte_read;
384   logic [2:0] sclk_bit_counter;
385
386
387
388   always_ff @(posedge sclk)
389     if (cs) begin
390         sclk_byte_read <= {sclk_byte_read[6:0], mosi};
391         sclk_bit_counter <= sclk_bit_counter + 3'd1;
392       end
393     else begin
394         sclk_byte_read <= 8'h00;
395         sclk_bit_counter <= 3'd0;
396       end
397
398   logic [2:0] bit_counter, previous_bit_counter;
399
400   always_ff @(posedge clk) begin
401       byte_read <= sclk_byte_read;
402       bit_counter <= sclk_bit_counter;
403       previous_bit_counter <= bit_counter;
404       gated_cs <= cs;
405     end
406
407
408   // when bit counter goes from 7 --> 0, make a pulse
409   assign byte_done = (previous_bit_counter == 3'd7 && bit_counter == 3'd0);
410 endmodule
411
412 module spi_controller(input logic clk, cs, byte_done,
413                input logic [7:0] byte_read,
414                output logic strip_num_en, offset_en, write_en, offset_incr,
flush);
415
416   typedef enum logic [1:0] {RESET, FLUSH, READ_OFFSET, READ_DATA} statetype;
417   statetype state, next_state;
418
419   always_ff @(posedge clk)
420     if (cs) state <= next_state;
421     else state <= RESET;
422
```

```
423  always_comb
424    if (byte_done)
425      case(state)
426        RESET: if(byte_read == 8'b01000110) next_state = FLUSH;
427             else if(byte_read < 8'd24) next_state = READ_OFFSET;
428             else next_state = RESET;
429        READ_OFFSET: next_state = READ_DATA;
430        READ_DATA: next_state = READ_DATA;
431        FLUSH: next_state = RESET;
432      endcase
433
434    else
435      next_state = state;
436
437  always_comb begin
438    if (byte_done)
439      case (next_state)
440
441        RESET: begin
442            strip_num_en = 1'b0;
443            offset_en = 1'b0;
444            write_en = 1'b0;
445            offset_incr = 1'b0;
446            flush = 1'b0;
447          end
448
449        FLUSH: begin
450            strip_num_en = 1'b0;
451            offset_en = 1'b0;
452            write_en = 1'b0;
453            offset_incr = 1'b0;
454            flush = 1'b1;
455          end
456
457        READ_OFFSET: begin
458            strip_num_en = 1'b1;
459            offset_en = 1'b0;
460            write_en = 1'b0;
461            offset_incr = 1'b0;
462            flush = 1'b0;
463          end
464
465        READ_DATA: begin
466            // if we are moving from READ_OFFSET --> READ_DATA
467            // we want to save the offset in a register
468            if (state == READ_OFFSET) begin
469                strip_num_en = 1'b0;
470                offset_en = 1'b1;
471                write_en = 1'b0;
472                offset_incr = 1'b0;
473                flush = 1'b0;
474              end
475            else begin
476                strip_num_en = 1'b0;
477                offset_en = 1'b1;
478                write_en = 1'b1;
479                offset_incr = 1'b1;
480                flush = 1'b0;
481              end
482          end
```

```
483
484         default: begin
485             strip_num_en = 1'bx;
486             offset_en = 1'bx;
487             write_en = 1'bx;
488             offset_incr = 1'bx;
489             flush = 1'bx;
490           end
491         endcase
492
493     else begin
494         strip_num_en = 1'b0;
495         offset_en = 1'b0;
496         write_en = 1'b0;
497         offset_incr = 1'b0;
498         flush = 1'b0;
499       end
500   end
501 endmodule
502
```