Weather Visualizing Cloud Lamp

Elizabeth Poss and Cordelia Stiff
E155 Final Project Report
December 8, 2017

Abstract: String lights of various shapes and sizes have become a staple of whimsical interior decorating. Our product aims to incorporate the functionality and excitement of home internet of things devices while still channeling that sense of whimsy. We have designed a cloud-shaped lamp which emulates current weather activity. Users control the lamp's power and brightness through a web server hosted by a Raspberry Pi, which also gets weather data from an API. This is communicated to an FPGA which decodes the weather data and uses SPI to control six LED strands. The cloud can display sunrise, sunset, and varying intensities of rain, snow, and lightning.

A. Introduction:

We created a lamp, shaped like a cloud, which displays real-time weather. Our finished prototype can both access current weather data, and display LEDs in patterns and colors which visualizes sunrise, sunset, and varying intensities of rain, snow, cloud cover, and lightning. The cloud is controlled via a web server, which allows the user to control the brightness of the lamp, and turn it on and off. A Raspberry Pi hosts the web server, and makes an API call to OpenWeatherMap's current weather API. Once the relevant weather data has been obtained and processed, it is transmitted to the FPGA via SPI. The FPGA uses a decoder to select the associated weather actions, and generates the data sent to the six LED strands, which are also controlled via SPI.

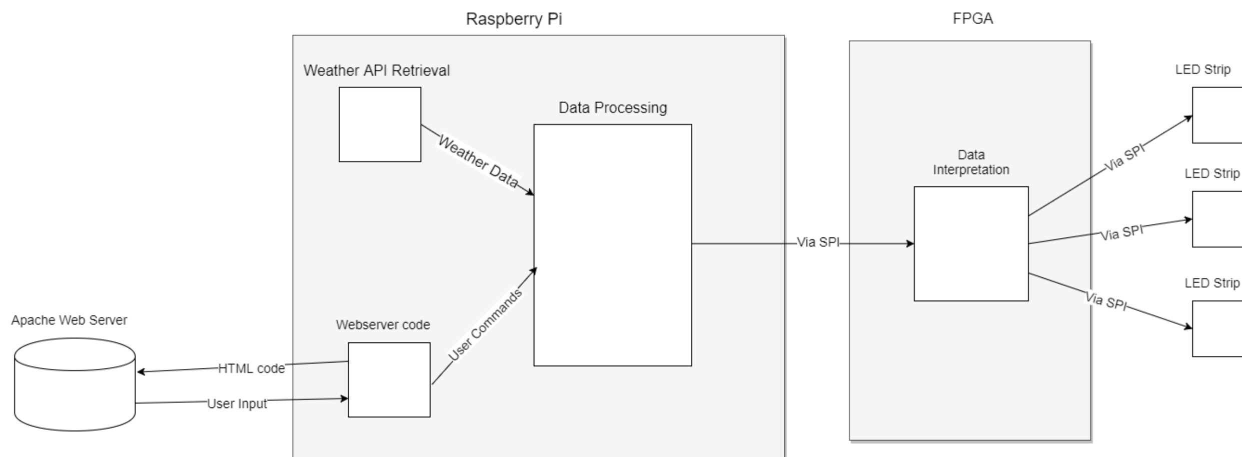

Figure One: Block Diagram of our system.

B. New Hardware:

Our new hardware element is the LED strands which we used to simulate weather. Our primary design choice concerned the choice of LEDs. For ease of comparison and ordering, we focused on LED options carried by Adafruit. LED strands were necessary for the rain/snow strands, so we chose to use strands in all parts of the lamp. We chose 30 LEDs per meter, the lowest LED density, since the size of the lamp and necessary length of the rain strands prioritized length over light density. This also allowed us to buy a greater length of LED strands. Finally, we chose DotStar LEDs over NeoPixels. The DotStar's strength in update speed, and use of SPI instead of PWM made us more comfortable using it, and made it preferable for things like our rain or lightning animations.

The DotStar LED strands we chose were composed of connected APA102C LEDs. These LEDs take a 5V and GND connection, as well as CI (clock in) and DI (data in). The

datasheet gives no specific constraints for the speed of the SPI transmission, and none were observed during testing beyond a rough max of ~1MHz, likely due to breadboard capacitance. The data sent was arranged in the following form: 4 bytes of 0, 4 bytes of information per LED, and an ending sequence of 4 bytes of 1.[1] Each LED took the format shown in Figure 2. The 5 bit global brightness value remains constant within a strand, but the RGB values are LED specific. An off LED pattern was designated with 0 values for brightness and RGB (32'hE0000000). This allows us to set specific LEDs on or off, to create patterns for our rain.

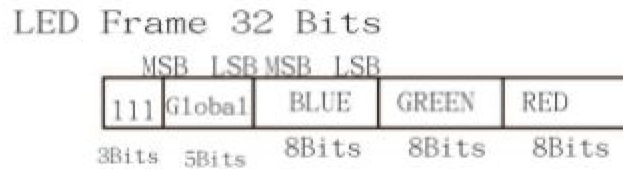

Figure 2: Data format for the APA102C. These 4 bytes would be necessary for each LED in the strand being controlled.[1]

In testing, it was observed that a data sequence with data for more LEDs than were present in the strand could be sent without error. However, if the information to control N LEDs was sent to a strand of length greater than N, then the N+1 LED in the strand would glow white, while the remaining LEDs would be dark as expected. Preliminary attempts showed no errors when the starting and ending sequences were omitted, but we chose to stick to the format recommended by the datasheet, and thus cannot guarantee that their omission would not lead to some bugs.

Ultimately, our SPI transmission was controlled by a second 'load' clock. This clock was timed to have a high period just longer than our longest necessary transmission time - in this case the time necessary to send 512 bits - and the data was retransmitted every clock cycle of load. This allowed us to generate rain by changing which bits were set on and off without having to call our SPI module multiple times.

The resources we consulted cautioned about power and current considerations. Fortunately, all of our strands at full brightness and white light never went above 2 Amps. However, for visualizations like lightning and rain, the current would swing dramatically as strands were turned on and off. While this was not a problem for a power supply, current and power considerations would be necessary for a project with more LEDs.

C. Schematics

See schematic below.

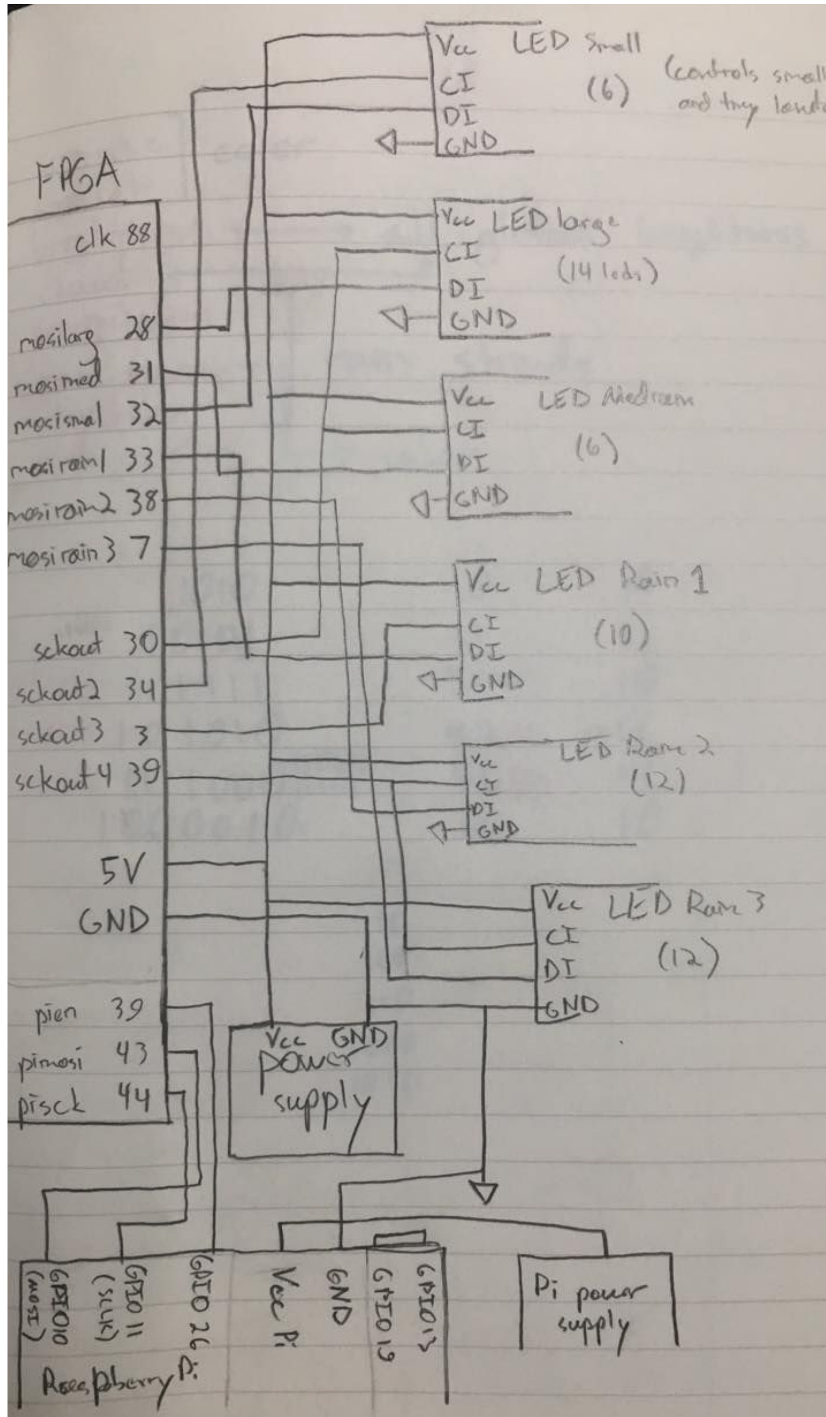

Figure 3: All elements were connected to the same ground. Everything but the Pi was connected to the same 5V power. Pins connected to the FPGA were multiples of the same Vcc/GNC/DI/CI format for the LED strands, and an Enable, MOSI, and SCLK for communication with the Pi.

D. Microcontroller Design:

The overall function of the Raspberry Pi was to control the settings for the lamp by retrieving the weather data from an API and hosting a web server to allow users to change basic settings of the lamp. There are three main modules on the Raspberry Pi, which I will now discuss in detail.

Section One - Web Server Code:

This section of code was used to host a web server, which allowed users to pick basic settings for the lamp. The webserver had two buttons, which turn the lamp on or off, and a text field, which lets user set a brightness for the cloud. We used Apache2 to host our web server, so most of the code for this was done in that format.

The web server relies on three main C functions. Two of the functions simply write a pin HIGH or LOW, respectively. These are used to turn the cloud on or off, as we will later check that pin in the main function. The other C file takes a brightness value from the text field using the GET HTTP protocol. It then processes the output of that field to get the integer brightness value. Finally, it writes that value out to a text file. This text file can later be accessed by our main function.

Section Two - API Call and Processing:

This section of the code does the processing to determine what series of bits should be sent to the FPGA. It is written in Python, and takes two inputs - a zip code, which determines the location that the API should check, and a user brightness. We will describe where these inputs come from in Section 3.

The first section of this code makes a request to the OpenWeatherMap API, with the chosen zip code. This will return a dictionary, which contains the weather conditions in that location.

We now need to process the bits. In the second part of this function, we check for several weather conditions: rain, snow, or lightning. This is done by checking the appropriate part of the dictionary for various weather conditions.[2] If we find any of these conditions, we then set the appropriate bits of our output. We then use the Python datetime library to get the current time. As the API uses Unix format for sunrise and sunset time, we also use the `datetime` library to convert to the format used in the rest of the `datetime` library. Using these times, we can check if we're within 30 minutes of either sunrise or sunset, and set the sunrise and sunset bits of our output.

Finally, we need to handle brightness. If the user has set the brightness to 0, then we use weather data to set brightness. The API call contains a section for cloudiness, so we can simply set the brightness bits to correspond to cloudiness. If the user picks a different brightness value, we first take the value modulo 31, so we don't overflow our five bit brightness value, and then set the brightness bits to this value.

Once we have the series of bits that we want to send, we simply convert them into an integer, and return the value.

Section Three - Main Function:

This is the function that does the main control for the cloud. We first initialize the various libraries we need, as well as the pins we will need to read to see if the cloud should be on or off. We then loop forever, so that the cloud is always on.

The first step we take is to get the brightness that the user has set. As we stated above in the description of the web server code, this is saved to a text file. Therefore, we simply need to open the file and read out the brightness.

We now need to check if the cloud should be on. In order to do this, we read the pin mentioned above in the web server section. If this pin is 0, we simply set the weather bits to 0 and do nothing else. Otherwise, we run a C function that gets the output of our Python function. We pass in a hard-coded zip code, as well as the user set brightness. We then check to make sure that the returned integer from our Python code is not equal to 0, which ensures that an error in the API call will not turn the cloud off. If we get a 0, we use the old bits - otherwise, we use the new bits calculated by the API call.

Finally, we need to send this data to the FPGA. We first write our SPI enable pin to 1. We then use the `EasyPIO.h` function `spiSendReceive16` to send the relevant bits to the FPGA, and then set the enable pin back to 0. We then delay for a brief period of time, as making successive API calls can cause the system to break. The loop then repeats forever, thus ensuring the cloud updates when the weather changes.

E. FPGA Design:

The FPGA served to accept weather data from the Pi, and control the LEDs. One SPI connection was made to the Pi, and six were made to the LED strands. Controlling the LEDs on the FPGA was necessary, as the rain strands had to be connected at only one end- this meant we would have needed an absolute minimum of three SPI

connections, which would be too many for the Pi. In addition, the rain and lightning visualizations required multiple clocks, which makes this better suited to the FPGA.

Section One- LED Control (SPI)

Six APA102C LED strands of various lengths were controlled by the FPGA. The data sent to each strand was determined by the format discussed in Section B. A data generation module took in a parameter length of the led strand, and a desired brightness and color, and output the full data string that would be sent to the strand. Rain was created using an `ledpattern` variable which marked the LED that would be on at a given instance. A series of ternary operators based on `ledpattern` created the SPI data string, which was sent to the SPI module. `ledpattern` was updated at the desired speed of the rain, and right shifted a single on LED in a loop through the bits of the variable. This simulated a raindrop falling down the strand. The color and speed of the rain could be modified with module inputs.

Section Two- Weather Decoder

A decoder converted weather signals like rain or sunset into LED control actions. The Pi communicates 16 bits, which include a five bit brightness value, single bits for sunrise, sunset, and rain/snow, and two bit precipitation and lightning values. The brightness value goes straight to the brightness input for the SPI generation modules, unless the precipitation value is zero, in which case brightness for the rain strands is zero, or if the lightning value is nonzero, in which case brightness for the lightning creating lantern changes in order to simulate lightning. This lightning creation logic uses a series of ANDed and XNOR'ed clock bits to create a flashing pattern, with varying

intensities based on the two lightning bits. The precipitation bits control speed, an input to the rain generation which controls the clock speed at which `ledpattern` updates. Finally, the rain/snow bit controls the color of the rain (blue for rain and white for snow), and sunrise and sunset change the colors of the lanterns from their default white to purples and oranges.

Section Three- Pi Communication (SPI)

The SPI connection with the Pi used a basic shift register. The Pi controlled an enable pin, and shifting only occurred while the enable pin was high. At the falling edge of the enable, the communicated value was shifted to the output of the function, ensuring that the decoder and LED control modules were controlled by a constant value.

F. Results:

We were able to complete our project successfully. Our lamp is covered in stuffing to make it appear like a cloud, and both the lantern strands and rain strands work correctly. We are able to generate all of the patterns we promised (rain, snow, and lightning), as well as various color patterns for sunrise and sunset. Finally, as promised, we created a web server that allows users to adjust brightness and turn the cloud on and off.

Figure 4: Completed Prototype. The cloud is displaying a sunset. The rain strands are not active. Hardware is visible in the upper right, protected from electrical shorts in a cardboard box.

The most difficult parts of our design were the two SPI connections. On the FPGA, we needed to both receive bits from the Raspberry Pi, and then turn these bits into the appropriate bits to send to the LED strips over SPI. While the basic SPI connection was simple to design, each side of this had particular needs that made the

process more difficult. For the Raspberry Pi SPI, we needed to ensure that we were getting the entire series of bits from the Pi before sending them on to our LED SPI module, so we didn't send bad patterns to the LED strips. This was achieved by attaching an "enable" pin from the Raspberry Pi, so we know exactly when the bits have begun to send, and when they've stopped sending. Ultimately the primary difficulty in testing this came from physical problems- our wires were too long, and created interference which caused the data read by the FPGA to be incorrect. Replacing the connections fixed this problem.

For the LED strands, we needed to create an SPI module that could take in basic inputs such as RGB color, brightness, and the length of the LED strand we were trying to use, and convert this into the proper series of bits. This was mainly done using block assignments and ternary operators, to assign the proper series of bits.

Our final prototype was consistent with our project proposal.

G. References

[1] Shiji Lighting. *APA102C Datasheet*. Adafruit. https://cdn-shop.adafruit.com/datasheets/APA102.pdf

[2] OpenWeatherMap. *Current Weather Data (API Documentation).* https://openweathermap.org/current

[3] S. Harris and D. Harris, *Digital Design and Computer Architecture: Arm edition*, Elsevier Science, 2015.

H. Parts List

Part Name	Link	Price per Unit	Units Needed	Total Cost
Paper Lanterns	ASIN:B01M63OEBF	$10.99	1	$10.99
Pillow Batting	Found in Makerspace	N/A	N/A	N/A

Dotstar 30 LED/m LED Strip	Adafruit: 2238	$19.95	2	$39.99

I. Code

spimoduletest.sv:

```
module paramcounter #(parameter N)
                                (input logic clk,
                                input logic reset,
                                output logic [N-1:0]q);
// basic parameterized counter
                always_ff @(posedge clk, posedge reset)
                        if (reset) q<= 0;
                        else q<=q+1;
endmodule


module paramspi #(parameter N)(input logic clk,
                                        input logic sck,
                                        output logic mosi,
                                        input logic [N-1:0]datain);
logic[N-1:0]p=0;//=160'b0;
logic[25:0]counter;
paramcounter #(26) loadclk(clk,1'b0,counter);
logic load;
// load clock calculated to take slightly longer than SPI for 512 bits
assign load = counter[16];
assign mosi = p[N-1];
always_ff @(posedge sck)
                if(load)
                        p<=datain;
                else p<= {p[N-2:0],1'b0};
// p is filled with zeros as they don’t change the LED actions
endmodule

// makes an led strand simulate rain
// rain strands are either 10 or 12
// commanding a strand of 10 leds with the information for 12 doesn’t cause errors, so assume a
length of 12 leds for all rain commands
// rain will probably always be white, but it can take any color input
module rain (input logic clk,
```

```
                                                input logic sck,
                                                input logic [4:0]globalbrightness,
                                                input logic [7:0]blue,
                                                input logic [7:0]green,
                                                input logic [7:0]red,
                                                input logic [1:0]speed,
                                                output logic mosi);
// create slow clock- constrained by time to control entire strand via spi (load clock)
logic [29:0]slockcount;
paramcounter #(30) slockmake(clk, 1'b0, slockcount);
logic slock;

assign slock = slockcount[22-speed];

// raininstance is the dataout sent to the led strand, ledpattern is the on/off pattern (12 bits, 1 on,
0 off)
logic [0:11]ledpattern;

logic[14*32-1:0]raininstance;
generateRainInstance eachrain(ledpattern, globalbrightness, blue, green, red, raininstance);

paramspi #(14*32) testled(clk,sck,mosi,raininstance);

// rotates led pattern- the two modules above will be called again every time ledpattern changes
always_ff @(posedge slock)
        begin
                if(ledpattern == 12'b000000000000)
                        ledpattern <= 12'b100000000000;
                else
                        ledpattern <= ledpattern >> 1;
        end
endmodule

// uses ledpattern to create the spi output for an individual moment of rain
module generateRainInstance(
                                                input logic [0:11]ledpattern,
                                                input logic [4:0]globalbrightness,
                                                input logic [7:0]blue,
                                                input logic [7:0]green,
                                                input logic [7:0]red,
                                                output logic[0:14*32-1]ledstring);
// constants based on datasheet
logic[31:0]startbits;
```

```
logic[31:0]endbits;
logic[31:0]ledbits;
logic[31:0]offled;
assign offled = 32'hE0000000;
assign startbits = 32'b0;
assign endbits = 32'hFFFFFFFF;
assign ledbits = {3'b111,globalbrightness,blue,green,red};

// assigns each part of ledstring bitwise (this is why the length is constant)
assign ledstring[0:31] = startbits;
assign ledstring[32*1:32*2-1] = (ledpattern[0] == 1)? ledbits : offled;
assign ledstring[32*2:32*3-1] = (ledpattern[1] == 1)? ledbits : offled;
assign ledstring[32*3:32*4-1] = (ledpattern[2] == 1)? ledbits : offled;
assign ledstring[32*4:32*5-1] = (ledpattern[3] == 1)? ledbits : offled;
assign ledstring[32*5:32*6-1] = (ledpattern[4] == 1)? ledbits : offled;
assign ledstring[32*6:32*7-1] = (ledpattern[5] == 1)? ledbits : offled;
assign ledstring[32*7:32*8-1] = (ledpattern[6] == 1)? ledbits : offled;
assign ledstring[32*8:32*9-1] = (ledpattern[7] == 1)? ledbits : offled;
assign ledstring[32*9:32*10-1] = (ledpattern[8] == 1)? ledbits : offled;
assign ledstring[32*10:32*11-1] = (ledpattern[9] == 1)? ledbits : offled;
assign ledstring[32*11:32*12-1] = (ledpattern[10] == 1)? ledbits : offled;
assign ledstring[32*12:32*13-1] = (ledpattern[11] == 1)? ledbits : offled;
assign ledstring[32*13:32*14-1] = endbits;

endmodule

// for communication between pi and fpga
module spi_slave_receive_only(input logic pien,
                                                                    input logic pisck,
//From master
                                                                    input logic
pimosi,//From master
                                                                    output logic [15:0]
data); // Data received
logic [15:0] q;
always_ff @(posedge pisck)
begin
	q<={q[14:0],pimosi};
end
// pien is an enable pin connected to the pi, it stays high for the duration of sending
always_ff @(negedge pien)
	data <= q;
endmodule
```

```
// create the SPI output to turn a led with numleds a singled color as input
module  valueGenOneColor#(parameter numleds)(
                                                input logic [4:0]globalbrightness,
                                                input logic [7:0]blue,
                                                input logic [7:0]green,
                                                input logic [7:0]red,
                                                output logic[0:((numleds+2)*32)-1]ledstring);
logic[31:0]startbits;
logic[31:0]endbits;
logic[31:0]ledbits;
assign startbits = 32'b0;
assign endbits = 32'hFFFFFFFF;
assign ledbits = {3'b111,globalbrightness,blue,green,red};

// {m{n}} replicates n m times
assign ledstring = {startbits, {numleds{ledbits}}, endbits};
endmodule


module spimoduletest(input logic clk,
                                                        input logic pimosi,
                                                        input logic pisck,
                                                        input logic pien,
                                                        output logic sckout,
                                                        output logic sckout2,
                                                        output logic sckout3,
                                                        output logic mosilarg,
                                                        output logic mosimed,
                                                        output logic mosismal,
                                                        output logic mosirain1,
                                                        output logic mosirain2,
                                                        output logic mosirain3);
// reset, enable, and slow clock for led SPIs
logic reset, sck;
assign reset = 1'b0;

// bit size of sck clock counter
parameter sckN = 30;
logic [sckN-1:0]counter;
paramcounter #(sckN) sckmake(clk, reset, counter);
assign sck = counter[6];
```

```
// sets clock output pins
assign sckout = sck;
assign sckout2 = sck;
assign sckout3 = sck;


// we have the following LED strands (followed by length)
// largest(14), medium(6), smalls(6), rain1(10), rain2(12), rain3(12)
// parameter constants of number of leds, followed by bit length
parameter larglen = 14;
parameter largb = ((larglen+2)*32);
parameter medlen = 6;
parameter medb = ((medlen+2)*32);
parameter smallen = 6;
parameter smalb = ((smallen+2)*32);
parameter rain1len = 10;
parameter rain1b = ((rain1len+2)*32);
parameter rain2len = 12;
parameter rain2b = ((rain2len+2)*32);
parameter rain3len = 12;
parameter rain3b = ((rain3len+2)*32);

logic [15:0] spiout;
//assign spiout = 16'b00_11111_1__0_11_1_11_11;
spi_slave_receive_only inittest(pien,pisck, pimosi,spiout);

// overall colors for rain and lantern, colors for each latern used for sunrise/set
logic [7:0]lred;
logic [7:0]lblue;
logic [7:0]lgreen;
logic [7:0]lred1;
logic [7:0]lblue1;
logic [7:0]lgreen1;
logic [7:0]lred2;
logic [7:0]lblue2;
logic [7:0]lgreen2;
logic [7:0]lred3;
logic [7:0]lblue3;
logic [7:0]lgreen3;
logic [7:0]rred;
logic [7:0]rblue;
logic [7:0]rgreen;
```

```
// rain/lanternbrightness used for cases where there's a difference between them
logic [4:0]globalbrightness,rainbrightness,lanternbrightness;
assign globalbrightness = spiout[13:9];

// speed controls rate of rain or snow
// lightning designates rate/existance of lightning
logic [1:0]speed, lightning;
assign speed = spiout[6:5];
assign lightning = spiout[3:2];

logic sunrise,sunset, rainsnow;
assign sunrise = spiout[15];
assign sunset = spiout[14];
//rain if 1, snow if 0
assign rainsnow = spiout[4];



always_ff @(posedge sck)
begin
	// we use logic for a series of counter bits to create a periodic section of lightning with
semi-random flashes within it
	// 01 is least, 10 is more lightning, 11 is more lightning, occuring twice as fast
	if(lightning == 2'b01)
		begin
			if((counter[29:26] == 4'b1111)&(counter[23]^counter[24]^counter[22]))
			begin
				lanternbrightness = 5'b00000;
			end
			else
				lanternbrightness = globalbrightness;
		end
	else if (lightning == 2'b10)
		begin
			if((counter[29:27] == 3'b111)&(counter[23]^counter[24]^counter[22]))
			begin
				lanternbrightness = 5'b00000;
			end
			else
				lanternbrightness = globalbrightness;
		end
	else if (lightning == 2'b11)
		begin
```

```
				if((counter[29:27] == 3'b111 | counter[29:27] ==
3'b011)&(counter[23]^counter[24]^counter[22]))
				begin
					lanternbrightness = 5'b00000;
				end
				else
					lanternbrightness = globalbrightness;
			end
		// brightness is unaffected if there is no lightning
		else lanternbrightness = globalbrightness;

		// color cases for lanterns- sunrise is orange, orange, pink, sunset is pink pink orange, all
white otherwise
		if(sunrise)
			begin
				lred1 = 8'hFF;
				lblue1 = 8'h00;
				lgreen1 = 8'h32;
				lred2 = 8'hFF;
				lblue2 = 8'hAA;
				lgreen2 = 8'h00;
				lred3 = 8'hFF;
				lblue3 = 8'h00;
				lgreen3 = 8'h52;
			end
		else if(sunset)
			begin
				lred1 = 8'hFF;
				lblue1 = 8'hAA;
				lgreen1 = 8'h00;
				lred2 = 8'hFF;
				lblue2 = 8'h00;
				lgreen2 = 8'h32;
				lred3 = 8'hFF;
				lblue3 = 8'h99;
				lgreen3 = 8'h00;
			end
		else if(!sunrise && !sunset)
			begin
				lred1 = 8'hFF;
				lblue1 = 8'hFF;
				lgreen1 = 8'hFF;
				lred2 = lred1;
```

```
                        lblue2 = lblue1;
                        lgreen2 = lgreen1;
                        lred3 = lred1;
                        lblue3 = lblue1;
                        lgreen3 = lgreen1;
                end

        // rain is blue because it's water, snow is white
        if(!rainsnow)
                begin
                        rred = 8'hFF;
                        rblue = 8'hFF;
                        rgreen = 8'hFF;
                end
        else if(rainsnow)
                begin
                        rred = 8'h00;
                        rblue = 8'hFF;
                        rgreen = 8'h00;
                end

        // if there is no rain, the brightness is set to zero which effectively turns them off
        if(speed==0)
                rainbrightness = 5'b00000;
        else if(speed != 0)
                rainbrightness = globalbrightness;
end

// assigns lanterns different dawn colors
// generate outputs
logic [largb-1:0]datainlarg;
valueGenOneColor #(larglen) orangetest(globalbrightness,lblue1,lgreen1,lred1,datainlarg);
logic [medb-1:0]datainmed;
valueGenOneColor #(medlen) bluetest(lanternbrightness,lblue2,lgreen2,lred2,datainmed);
logic [smalb-1:0]datainsmal;
valueGenOneColor #(smallen) pinktest(globalbrightness,lblue3,lgreen3,lred3,datainsmal);


// do the spi
paramspi #(largb) bigstrand(clk,sck,mosilarg,datainlarg);
paramspi #(medb) medstrand(clk,sck,mosimed,datainmed);
paramspi #(smalb) smalstrand(clk,sck,mosismal,datainsmal);
```

```
// controls three rain strands
rain createrain(clk,sck, rainbrightness,rblue,rgreen,rred,speed,mosirain1);
rain rain2constructor(clk,sck, rainbrightness,rblue,rgreen,rred,speed,mosirain2);
rain rain3constructor(clk,sck, rainbrightness,rblue,rgreen,rred,speed,mosirain3);

endmodule
```

***apiCallSpi.c*:**

```
#include <Python.h>
#include "EasyPIO.h"

// Code from Python documentation with slight modifications
// Gets integer output from Python code

int getWeatherInt(char* zipCode, int brightness)
{
        // Make c actually import the pythonpath
setenv("PYTHONPATH", ".", 1);
   // Create the arguments
   int argc = 5;
   char** argv = (char**)malloc(sizeof(char*)*argc);
   argv[0] = "./apiCall";
   argv[1] = "apiCall";
   argv[2] = "mainFunc";
   argv[3] = zipCode;
        char str[10];
   sprintf(str, "%d", brightness);
   argv[4] = str;

   PyObject *pName, *pModule, *pDict, *pFunc;
   PyObject *pArgs, *pValue;
   int i;

   // Create a variable to store our output
   int outputVal = 0;

   if (argc < 3) {
      fprintf(stderr,"Usage: call pythonfile funcname [args]\n");
      return 1;
   }

    Py_Initialize();
   pName = PyUnicode_FromString(argv[1]);
   /* Error checking of pName left out */
   pModule = PyImport_Import(pName);
   Py_DECREF(pName);

   if (pModule != NULL) {
      pFunc = PyObject_GetAttrString(pModule, argv[2]);
```

```
        /* pFunc is a new reference */

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyLong_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument\n");
                    return 1;
                }
                /* pValue reference stolen here: */
                PyTuple_SetItem(pArgs, i, pValue);
            }
            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);
            if (pValue != NULL) {
                outputVal = PyLong_AsLong(pValue);
                Py_DECREF(pValue);
            }
            else {
                Py_DECREF(pFunc);
                Py_DECREF(pModule);
                PyErr_Print();
                fprintf(stderr,"Call failed\n");
                return 1;
            }
        }
        else {
            if (PyErr_Occurred())
                PyErr_Print();
            fprintf(stderr, "Cannot find function \"%s\"\n", argv[2]);
        }
        Py_XDECREF(pFunc);
        Py_DECREF(pModule);
    }
    else {
        PyErr_Print();
        fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
        return 1;
    }
    return outputVal;
```

```
}

void delayMinutes(int numMinutes){
   /*
      Takes in a number of minutes, and delays for that long
      Relies on underlying code in EasyPIO
   */

   // delay in milliseconds
   int delayInMillis = 6000 * numMinutes;

   delayMillis(delayInMillis);
}

int getUserBrightness(void){

         /*
         Opens a txt file to see what the user has set the brightness to
         Returns this as an integer
         */
         FILE* brightnessFile;
         char buff[255];

         brightnessFile = fopen("brightness/brightness.txt", "r");
         if (brightnessFile != NULL)
         fscanf(brightnessFile, "%s", buff);
         else
{
         printf("file not opening");
         return 0; }
         return atoi(buff);
}



int main(){
   /*

   Runs a timer. Every so often, checks the weather, and then sends the bits over SPI

   */
   // We only need to initialize EasyPIO and SPI once
   pioInit();
```

```
    spiInit(250, 0);
    printf("Starting program \n");


   // Set up pins we need for SPI
   pinMode(19, INPUT);
   pinMode(21, OUTPUT);
int i = 0;
    // While loop forever, because we want to constantly be checking
   // Shorter loop interval for demo
    while(i < 10){
         // Get the weather bits
        printf("loop ran\n");
        int userBrightness = getUserBrightness();
        printf("User brightness is %d\n", userBrightness);
        int weatherBitVal;
        // If the light is off, don't  get the weather
        if (digitalRead(19) == 0)
                weatherBitVal  = 0;
        else{
                // Get the weather bits
                int weatherBits = getWeatherInt("91711", userBrightness);
                // If the weather bits are 0, don't change them
                // Don't want to turn it off because of API errors
                if (weatherBits != 0)
                weatherBitVal = weatherBits;
        /*

        DEMO MODE GOES HERE

        loop 0 = sunrise
        loop 1 = sunset
        loop 2 = low speed lightning and low speed rain
        loop 3 = high speed lightning and high speed snow
        AFTER LOOP 3 USE LIVE WEATHER DATA
        loop 4 = user defined brightness
        loop 5 = automatic brightness
        loop 6 = normal weather (turn the cloud off)
        loop 7 = normal weather (turn the cloud back on)

        */
        if (i == 0){
                weatherBitVal = 48899;
```

```
        }
        else if (i == 1)
                weatherBitVal = 32515;
        else if (i == 2)
                weatherBitVal = 16167;
        else if (i == 3)
                weatherBitVal = 16255;

}
      printf("Bits have integer value of %d \n", weatherBitVal);
        printf("%d \n", weatherBitVal);
        i++;

        //Write our SPI enable pin high
        digitalWrite(21, 1);
        // Send the relevant data
        spiSendReceive16(weatherBitVal);
        // Write the SPI enable pin low
        digitalWrite(21, 0);
        // Wait for some time before checking again
        printf("Delaying");
        delayMinutes(3);
        printf("%d \n", i);

}
// Stop the python interpreter
Py_Finalize();
printf("for loop done \n");
}
```

cloudBrightness.c:

```
#include <stdio.h>

int main(void){

	// Print the HTML header
	printf("%s%c%c\n", "Content-Type:text/html;charset=iso-8859-1",13,10);

	// Get our brightness value from the QUERY_STRING value
	const char* brightnessValue = getenv("QUERY_STRING");
	// Get a pointer to the equal sign
	char* arg  = strchr(brightnessValue, '=');

	// Check if the value is null
	if (brightnessValue == NULL){
		printf("Sorry, brightness value cannot be read");
	}


	// If not, write the brightness value out to a text file
	else{
		FILE* brightnessFile;
		const char* filename =
"/home/pi/Desktop/FinalProject/microPsFinalProject/brightness/brightness.txt";
		const char* mode = "w";
		brightnessFile = fopen(filename, mode);

		// Write out the brightness
		if (brightnessFile != NULL){
			// Only  write characters after the equal sign
			arg++;
			printf("%c\n", *arg);
			while (*arg)
			{
				printf("%c\n", *arg);
				fputc(*arg, brightnessFile);
				++arg;
}
			// Close the file
			fclose(brightnessFile);
			// Redirect back to the homepage
			printf("<META HTTP-EQUIV=\"Refresh\"
```

```
CONTENT=\"0;url=/cloud.html\">");
                    }
                    // If we can't open the file, display an error
                    else{
                              printf("error saving to file");
}}
          return 0;

}
```

cloudOff.c:

```
#include "EasyPIO.h"

int main(void){

	// Initialize EasyPIO
	pioInit();

	// Set up pin 21 to write
	pinMode(13, OUTPUT);

	// Write low  to the pin
	digitalWrite(13, 0);

	// Print the HTML header
	printf("%s%c%c\n", "Content-Type:text/html;charset=iso-8859-1",13,10);

	// Redirect back to main page
	printf("<META HHTP-EQUIV=\"Refresh\" CONTENT=\"0;url=/cloud.html\">");

	return 0;

}
```

cloudOn.c:

```
#include "EasyPIO.h"

int main(void){

	// Initialize EasyPIO
	pioInit();

	// Set up pin 21 to write
	pinMode(13, OUTPUT);

	// Write high to the pin
	digitalWrite(13, 1);

	// Print the HTML header
	printf("%s%c%c\n", "Content-Type:text/html;charset=iso-8859-1",13,10);

	// Redirect back to main page
	printf("<META HHTP-EQUIV=\"Refresh\" CONTENT=\"0;url=/cloud.html\">");

	return 0;

}
```

apiCall.py:

```
import requests
from tzwhere import tzwhere
from datetime import datetime, timedelta
from pytz import timezone
import pytz

API_KEY = '4f6f8f30f593ed64cec14b81dd480eb2'


def mainFunc(zipCode, userBrightness):
	'''
		Takes in a zipcode, and returns the correct sequence of bits
	'''

	# get the weather dictionary
	weatherDict = getWeather(zipCode)

	# Run the final
	return setWeatherBits(weatherDict, userBrightness)


def getWeather(zipCode):

	zipCode = int(zipCode)

	# Create the parameters
	payload = {'zip': zipCode, 'APPID': API_KEY, 'units': "Imperial"}

	# Do the get request
	r = requests.get('http://api.openweathermap.org/data/2.5/weather', params=payload)

	# Conver the string version of dictionary to an actual dictionary
	dictionary = eval(r.text)

	print(dictionary)
	# return the weather text
	return dictionary


def getCurrentTime(coordinates):
```

```
	# First, get the timezone in a location

	# set up tzwhere to get timezones
	tz = tzwhere.tzwhere()

	# Get the latitude and longitude out of the dictionary
	latitude = coordinates["lat"]
	longitude = coordinates["lon"]

	# Calculate the time zone
	tzResult = tz.tzNameAt(latitude, longitude)

	# Get the current UTC time
	currentTime = datetime.now(timezone(tzResult))

	currentTime = currentTime.replace(tzinfo = None)

	return currentTime

def setWeatherBits(weatherDictionary, userBrightness):
	'''
		Takes in the weather information and sets the bits correctly
	'''
	# Set everything to 0 except our padding bits

	# First two bits = sunrise or sunset
	# Next 5 = brightness
	# Last = padding
	brightnessBits = [0, 0, 0, 0, 0, 0, 0, 1]

	# First bit is 0 cause we don't have anything to put here
	# Second and third are precipitation (none/some/more/hella)
	# 1 bit if precipitation is rain or snow (rain = 0)
	# 2 bits for lightning (none/some/more/hella)
	# 2 bits of padding
	weatherBits = [0, 0, 0, 0, 0, 0, 1, 1]

	# Entirely padding
	paddingBits = [0, 0, 0, 0, 0, 0, 0, 0]

	# Check to make sure we actually have data. If we don't, just return 0
	try:
		weatherDictionary["coord"]
```

```
        except KeyError:
                return 0

        # Get the time
#       currentTime = getCurrentTime(weatherDictionary["coord"])
        currentTime = datetime.now()

        #########################
        #  SUNSET/SUNRISE/TIME  #
        #########################

        # Get the time for sunrise and sunset
        sunrise =  datetime.fromtimestamp(
      weatherDictionary['sys']['sunrise'])
        sunset =  datetime.fromtimestamp(
      weatherDictionary['sys']['sunset'])

        # Calculate the amount of time until sunrise and sunset
        timeToSunrise = abs(currentTime - sunrise)
        timeToSunset = abs(currentTime - sunset)

        # Create a time delta of 30 minutes
        previousTimeDelta = timedelta(minutes = 30)


        # Check if we are within 30 minutes of the sunrise
        if currentTime >= sunrise - previousTimeDelta and currentTime <= sunrise +
previousTimeDelta:
                # If we are, set sunrise bits to 1
                brightnessBits[0] = 1

        elif currentTime >= sunset - previousTimeDelta and currentTime <= sunset +
previousTimeDelta:
                brightnessBits[1] = 1


        #####################################
        #                RAIN/LIGHTNING/CLOUDS          #
        #####################################

        # Check weather conditions
        for weatherCond in weatherDictionary['weather']:
```

```
		description = weatherCond['description']

		# RAIN
		if 'Rain' in weatherCond['main']:
			# set weather bit to 0
			weatherBits[3] = 0
			# Check how much rain there is
			if description == "light rain" or description == "light intensity shower rain":
				weatherBits[1:3] = [0, 1]
			elif description == "moderate rain" or description == "shower rain":
				weatherBits[1:3] = [1, 0]
			else:
				weatherBits[1:3] = [1, 1]
		# SNOW
		if 'Snow' in weatherCond['main']:
			# set weather bits to 1
			weatherBits[3] = 1
			if description == "light snow" or description == "light rain and snow" or
description == "light shower snow":
				weatherBits[1:3] = [0, 1]
			elif description == "snow" or description == "rain and snow" or description
== "shower snow":
				weatherBits[1:3] = [1, 0]
			else:
				weatherBits[1:3] = [1, 1]
		# LIGHTNING
		if 'Thunderstorm' in weatherCond['main']:
			if description == "light thunderstorm" or description == "thunderstorm with
light rain" or description == "thunderstorm with light drizzle":
				weatherBits[1:3] = [0, 1]
			elif description == "thunderstorm with rain" or description ==
"thunderstorm" or description == "thunderstorm with drizzle":
				weatherBits[1:3] = [1, 0]
			else:
				weatherBits[1:3] = [1, 1]


	###########################
	#		BRIGHTNESS			#
	###########################

	# Get the cloud percentage
	# 1 = no clouds (i think)
```

```
        cloudiness = weatherDictionary['clouds']['all']
        # Check if the user input a brightness. Only get cloud info if user brightness is 0
        if userBrightness == 0:
                if cloudiness < 25:
                        brightnessBits[2:-1] = [1, 1, 1, 1, 1]
                elif cloudiness < 50:
                        brightnessBits[2:-1] = [1, 1, 0, 0 ,0]
                elif cloudiness < 75:
                        brightnessBits[2:-1] = [1, 0, 0, 0, 0]
                else:
                        brightnessBits[2:-1] = [0, 1, 0, 0, 0]
        # Otherwise, convert to binary
        else:
                # Take user input mod 32 (since we don't want input > 32)
                userBrightness = userBrightness % 31

                # Convert to binary
                userBrightnessBinary = "{0:05b}".format(userBrightness)

                # Make binary string into list
                userBrightnessList = list(map(int, userBrightnessBinary))

                # Add user brightness into array
                brightnessBits[2:-1] = userBrightnessList

                print(userBrightnessList)
        finalArray = brightnessBits + weatherBits

        print(finalArray)

        # Convert the bits to an integer, and send that
        intToReturn = convertBitsToInt(finalArray)


        return intToReturn


def convertBitsToInt(bitArray):
        '''
                Takes in a array of bits and converts it to a int
        '''
        finalResult = 0
```

```
# Flip the list because it's in MSB order
bitArray.reverse()

# Loop through the array
for i in range(len(bitArray)):

	# Add the bit * 2^i to our final result
	finalResult += ((2**i) * bitArray[i])

return finalResult
```