Project Report

Marz Barnes, Max Castro

Abstract

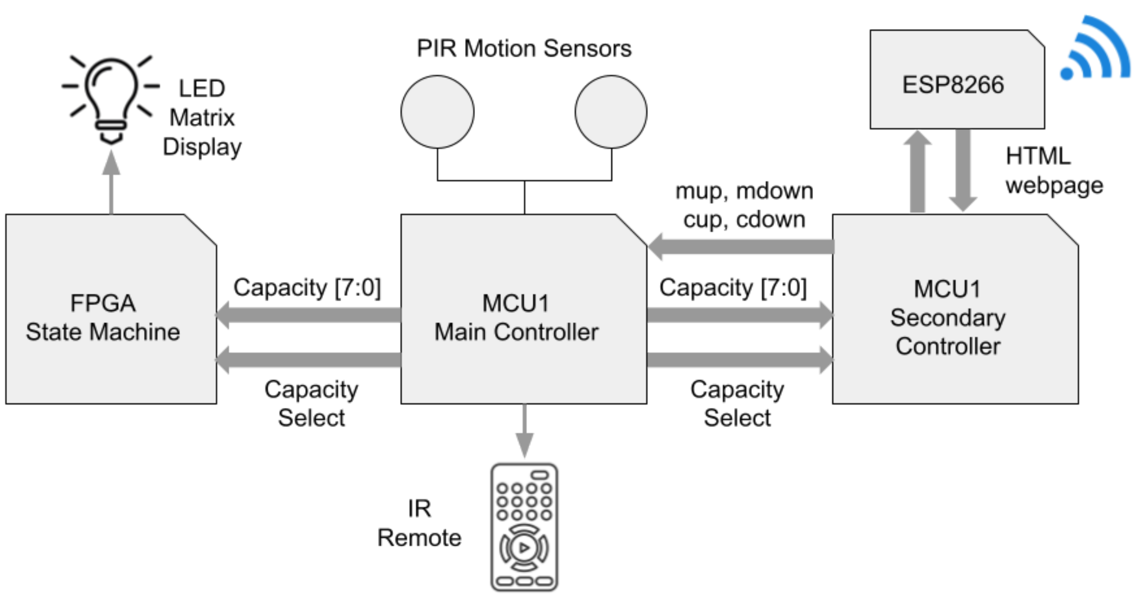
A microprocessor based system was designed by the team to track the current and maximum occupancy of a room. The system is intended to simplify managing more restrictive capacity constraints imposed by the COVID-19 pandemic. The delivered system counts people as they enter and exit through a single doorway with reasonable accuracy, displays the current estimated and maximum occupancy, and allows the user to manually make changes to adjust for errors or to adjust the maximum occupancy.

Introduction

Motivation

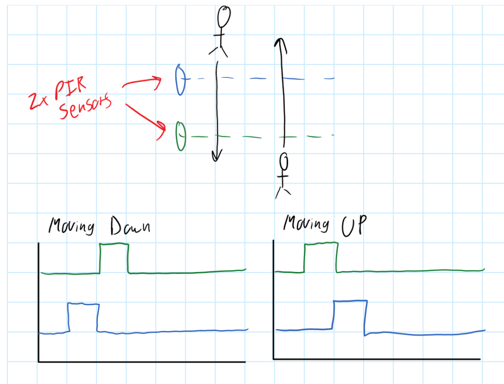
Managing capacity in a room is more important than ever now. Instead of having to staff someone to count everyone that walks in and out, we have devised a system that counts people as they enter and leave through the same doorway, eliminating the need to have a person staffed to manually count attendees.

Block Diagram



Overview

The system uses two passive infrared (PIR) motion sensors to determine whether a person is entering or exiting, as shown in Figure 1. Depending on whether the person is entering or exiting, one of the two sensors will fire first.

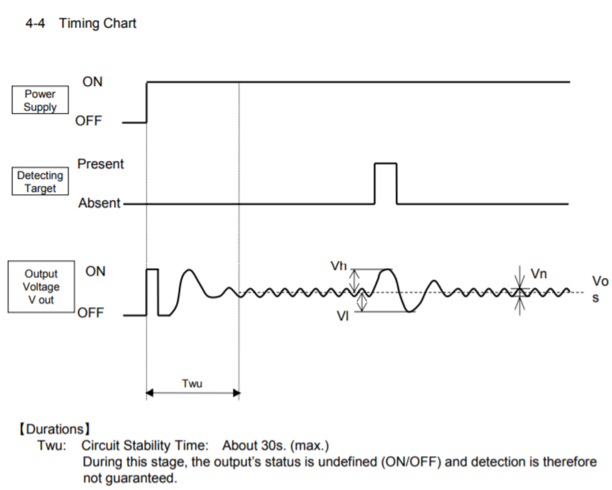


The MCU keeps track of the number of people that are inside the room and displays the current count on an LCD matrix. The user may use an infrared (IR) remote to manually adjust values if needed. Additionally, the user may set the maximum occupancy, and the LED matrix will indicate when capacity has been reached. The user may also use a website powered by an ESP8266 wi-fi module to view and edit the current occupancy and maximum occupancy.

New Hardware

PIR Sensors

Passive infrared sensors sense motion by detecting infrared radiation from body heat. The output of the sensor changes as the amount of infrared light it detects changes. We used two Panasonic EKMC2601111K sensors. These sensors have just three pins: 3.3V input, GND, and an analog output pin [1]. Example output from the datasheet is shown in below:

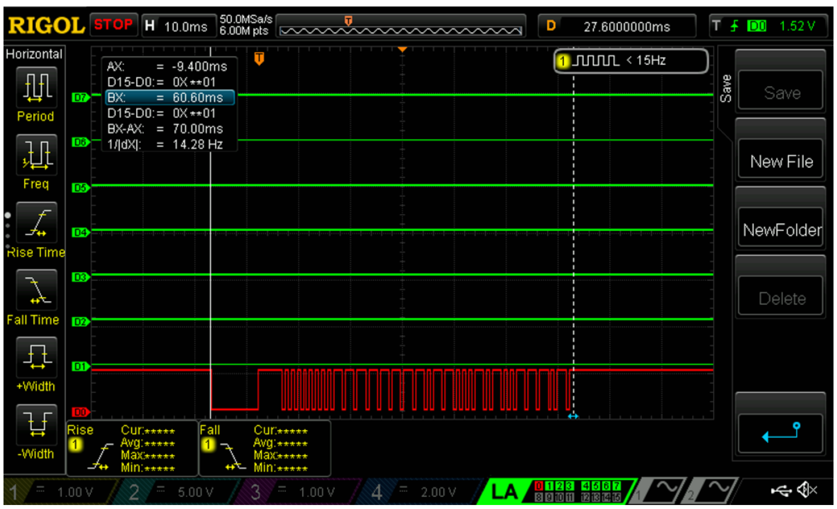


The MCU uses its ADC to detect the voltage level output by the two PIR sensors. When the voltage exceeds a certain value, the MCU interprets this as motion being detected.

Our testing found that a person walking by the PIR sensor would always cause the sensor to rail out several times in both directions and that it would take several seconds for the sensor output to return to its default value. To make it easier for the MCU to detect a person and to reduce the amount of time needed for the sensor to return to its original value, we covered most of the surface of the PIR sensors to restrict the amount of infrared light detected. With the coverings, when a person walked by the sensors, they would only output a single peak (as opposed to multiple rail-outs) and would quickly return to their starting value.

IR Remote and Receiver

We used an Arduino IR remote and receiver to communicate with the MCU. The IR receiver module has three pins: 3.3V power, GND, and a digital output pin. The remote outputs a binary sequence of bits via its light source whenever a button is pressed. The MCU then receives this sequence and decodes it to determine which button was pressed. For example, this is the signal received for one of the buttons:

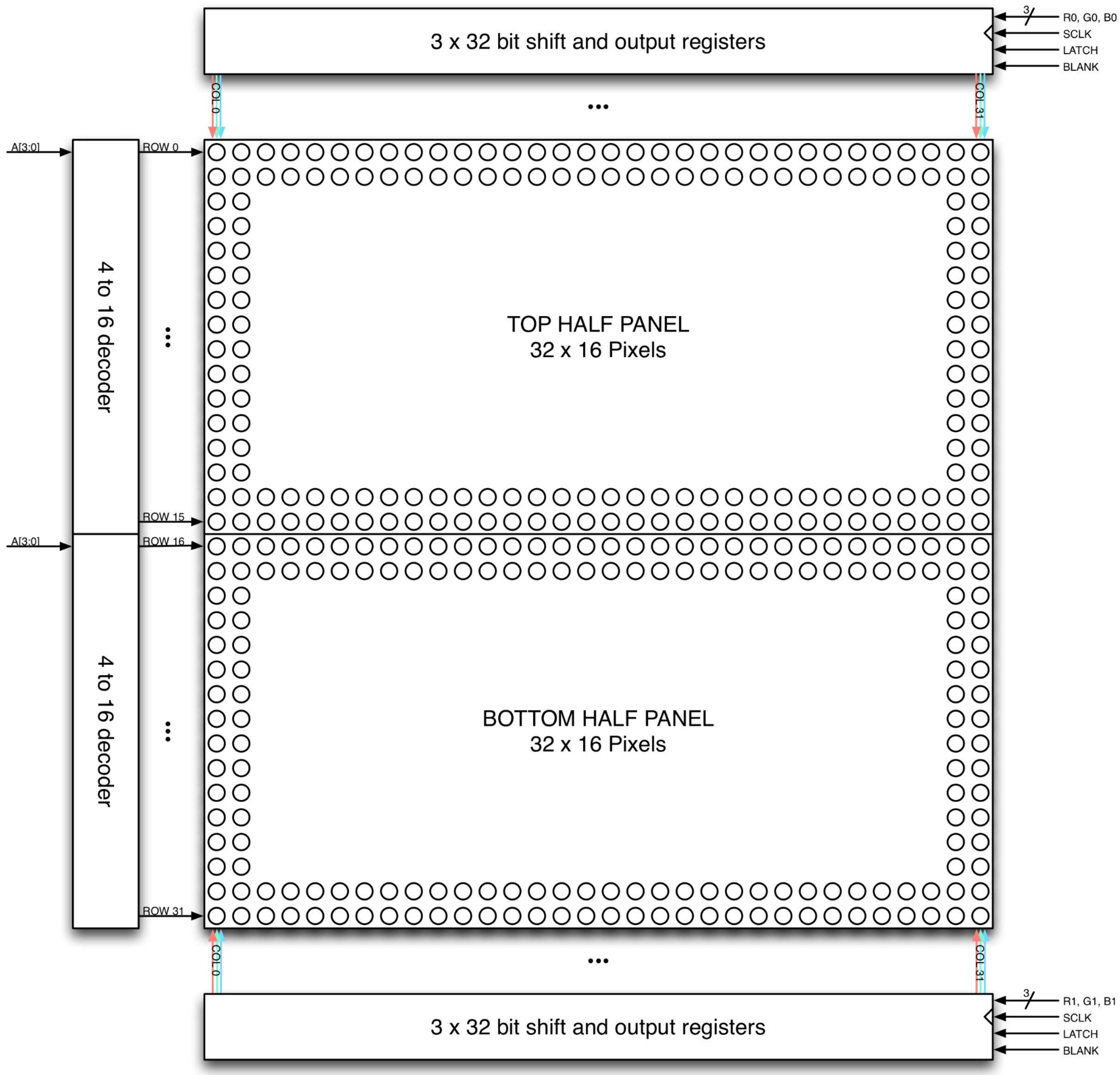


To receive and decode button presses, the MCU constantly reads the value of the IR module output pin. The signal rests at a high value, so once the MCU detects a low value, it begins reading the signal. The MCU uses a delay function (which itself utilizes on-board timers) to wait until the proper time to read each bit from the IR receiver module.

LED Matrix

The 32x32 LED Matrix was a difficult piece of hardware to figure out. We used the hardware from a previous MicroPs group, which was helpful because in their report, they found good resources that described how the hardware worked and also one way to interface with it [2].

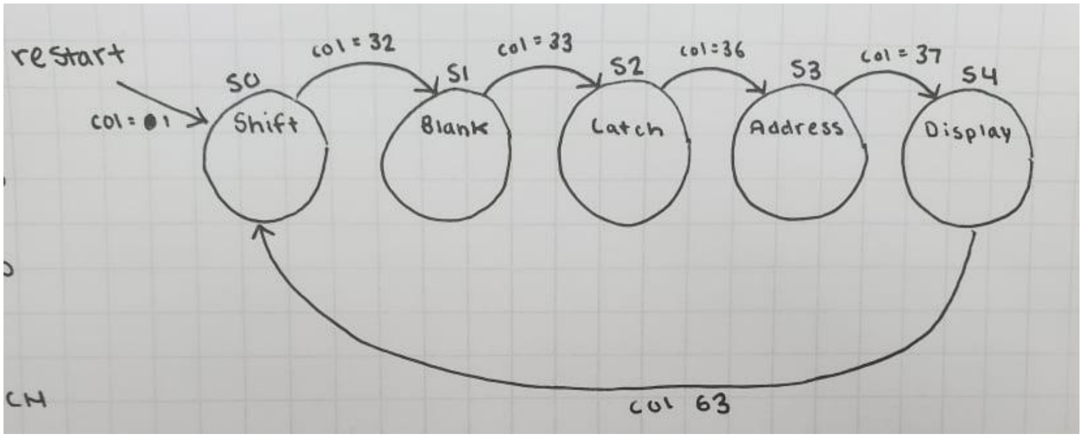
Essentially, there are two data buses that control the top and bottom half of the display: (R0, G0, B0) and (R1, G1, B1), and one row address bus, A[3:0], that controls which row to be displayed. The display will light up two rows at a time, for example Row 0 and Row 16, on the top and bottom half of the displays. Below shows a schematic of this from the Glen Akin's wikipedia page [3]:



To drive the display, a very specific sequence of events needs to be triggered, for a single row to be lit up. This sequence is outlined (again) by the Glen Akin's tutorial [3]:

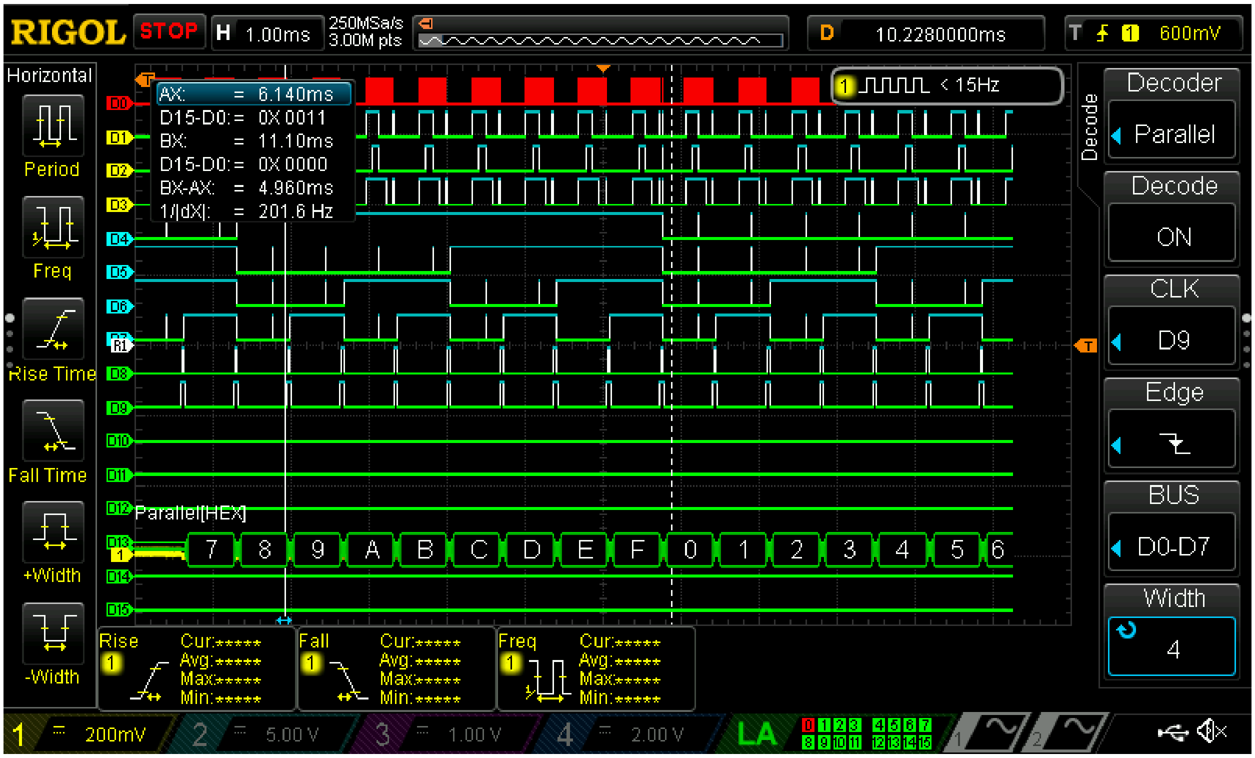
1. Shift in the 32 bits of RGB data using the R0, G0, B0, R1, G1, B1 buses and SCLK.
2. Assert BLANK = 1 to blank the display.
3. LATCH the contents of the column driver's shift registers by switching on and off LATCH.
4. Set A[3:0] to the row that is next to display.
5. Deassert BLANK = 0 to display the row.

So, we designed an FSM to complete all of these steps in sequence, and then repeat for each row to be displayed. A diagram for which is shown below:



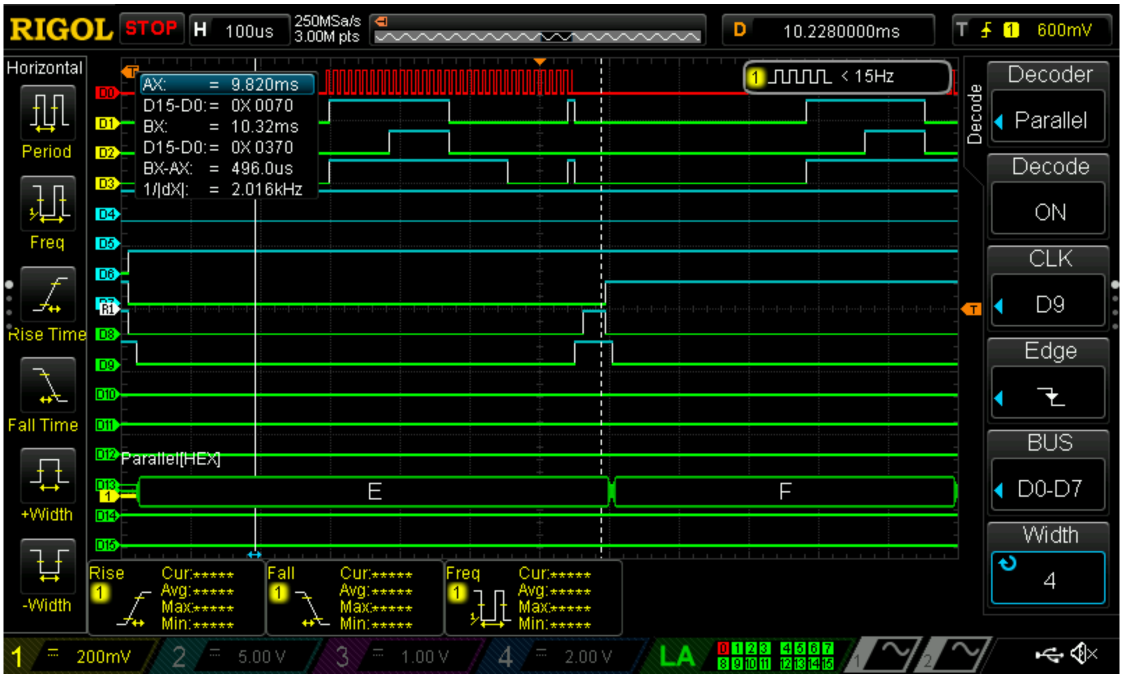
Next, we chose an appropriate clock frequency. To run the display, the rows need to be time multiplexed at 1/16th the total frequency, since only one set of rows can be driven at a time. According to the reference document, the display should update every 100-200 seconds to avoid flickering. Meaning that each row should be displayed at a frequency of 1.6-3.2kHz. Since each run of the FSM is 64 clk cycles, this means that our target clk frequency should be around 100-200kHz. Since our internal clk is 12MHz, we downsample this by 2^6 = 64, to achieve a desired clk frequency of 187.5kHz.

We hooked up the pins to our display, and debugged the system by checking the pins with the logic analyzer, eventually getting the following outputs shown by the figures below. There is a bit of noise (probably from parasitics) but it doesn’t seem to mess with the display.



Note that D0 is SCLK, D1-D3 are R0, G0, and B0, D4-D7 are A3, A2, A1, and A0 (also shown in HEX by the Parallel Decoder), D8 is LATCH, and D9 is BLANK. We see the large cycle of iterating over each row address A[3:0] from 1 to 16. As shown below, SCLK should only be sent while the RGB data transfer is happening.

Zooming in on a single row cycle, we see more visibly the SCLK signal, the data transfer for the first 32 cycles corresponding to each column on lines D1 through D3 (R0, G0, B0). We also see the order of blank → latch → address → unblank is being followed properly by the lines D4-D7 (A[3:0]), D8 (LATCH), and D9 (BLANK).



After getting the display to work for a static and repeating matrix, we updated the code to display a matrix stored on the FPGA and updated with the values for maximum and current capacity. At the start of each display refresh (col==0, row==0), we introduced a bit of next state logic to read and save the current updated value sent by the MCU for either maximum capacity or current capacity and update the display accordingly.

We then create multiple layers of combinational logic to decode this into the row of RGB data that we want to send to the state machine. The lowest level being a digit decoder that takes a single digit and outputs a 5x5 bitmap for that character. The level above that being the converter that takes in capacity [7:0] and decodes an entire 5x32 bitmap displaying that 3 digit number. The top level being a brick of combinational logic that stores the entire 32x32 bitmap and determines the current RGB values to display from the information on the current row being displayed.

Schematic/Wiring Diagram

Results

The results of this capacity monitoring system were very successful. Setting up the PIR motion sensors on a table, and walking in front of them in both directions, they were able to read changes in capacity to about a 70-80% accuracy. And these numbers were appropriately sent and displayed on both the LED matrix and on the website hosted by the ESP8266. Additionally, user inputs that change in maximum and current capacity sent in by both the IR remote and the website hosted by the ESP8266 were able to update both the LED matrix and on the website. Linked below is a brief video showing this system in action:

https://drive.google.com/file/d/1otaWAF4-AMS8CwZeu324x4Ngav3lktR3/view?usp=sharing

References

[1] PIR sensor datasheet: https://www.farnell.com/datasheets/2617518.pdf

[2] David Sobek and Jerry Liang, *Final Project Report: Bead Maze with LED Matrix and Accelerometer*, E155 Final Report, 2019. http://pages.hmc.edu/harris/class/e155/projects19/Sobek_Liang.pdf

[3] Adkins, Glen. “RGB LED Panel Driver Tutorial.” RGB LED Panel Driver Tutorial, 2014, https://bikerglen.com/projects/lighting/led-panel-1up/.

Bill of Materials

Name	**Part No.**	**Cost/Unit**	**Quantity**	**Total Cost**
MCU	STM32F401RE	$13.83	2	$27.66
FPGA	MAX1000	$26.66	1	$26.66
PIR Motion Sensors	EKMC2601111K	$5.54	2	$11.08
LED Matrix	[1]		1	
IR Remote and Receiver	[2]		1	
Wi-Fi Module	[3]		1	

[1] Borrowed from the E155 supply cabinet. A previous project purchased this matrix.
[2] Leftover from an Elegoo UNO R3 Super Starter Kit (SP20 E80 Lab Kit)
[3] Borrowed from the E155 supply cabinet.

C Code

Main MCU (Runs all devices except the ESP8266)

```
////////////////////////////////////////////////
// Main
////////////////////////////////////////////////
```

```
#include "STM32F401RE.h"
#include <stdbool.h>
#include <string.h>

int main(void) {

  configureFlash();
  configureClock();

  // Enable GPIOA&C and TIM2 clock
  RCC->AHB1ENR.GPIOAEN = 1;
  RCC->AHB1ENR.GPIOBEN = 1;
  RCC->AHB1ENR.GPIOCEN = 1;
  uint32_t* RCCPtr = ((uint32_t *)0x40023840);
  *RCCPtr |= (1 << 0); // Enable timer2



  initTIM(TIM2);
  initADC();

  // // LED pins
  pinMode(GPIOA, 0, GPIO_OUTPUT); // Down LED
  pinMode(GPIOA, 1, GPIO_OUTPUT); // Up LED
  // pinMode(GPIOA, 6, GPIO_OUTPUT); //errors LED
  // pinMode(GPIOA, 4, GPIO_INPUT); // IR out
  pinMode(GPIOB, 0, GPIO_OUTPUT); // Test signal. Tells us when the MCU is
  //                                 // reading the IR output.

  // LED Matrix Pins
    //PA3 (2^7)
    //PA2
    //PA10
    //PB3
    //PB5
    //PB4
    //PB10
    //PA8 (least sig)

    //PA9 (select)
```

```
  pinMode(GPIOA, 3, GPIO_OUTPUT);
  pinMode(GPIOA, 2, GPIO_OUTPUT);
  pinMode(GPIOA, 10, GPIO_OUTPUT);
  pinMode(GPIOB, 3, GPIO_OUTPUT);
  pinMode(GPIOB, 5, GPIO_OUTPUT);
  pinMode(GPIOB, 4, GPIO_OUTPUT);
  pinMode(GPIOB, 10, GPIO_OUTPUT);
  pinMode(GPIOA, 8, GPIO_OUTPUT);

  pinMode(GPIOA, 9, GPIO_OUTPUT);

  digitalWrite(GPIOB, 0, 0);

  // Min and Max ADC output, for debugging
  int Max1 = 0;
  int Max2 = 0;
  int Min1 = 9999;
  int Min2 = 9999;

  // Interior of room and exterior sensors triggered.
  bool extSensorThisTime = 0;
  bool intSensorThisTime = 0;

  bool extSensorLastTime = 0;
  bool intSensorLastTime = 0;

  // People entering and leaving for debugging
  int peopleIn = 0;
  int peopleOut = 0;

  // FSM for determining if someone is entering or leaving. # people in
room. # error inputs
  // the MCU doesn't understand
  int state = 0;
  int people = 0;
  int errors = 0;

  // Default capacity
  int capacity = 10;
```

```
  // Timeout timer if someone activates one PIR without
  // activating the other
  long timer = -1;

  // Whether the people or occupancy needs to be updated
  bool peopleChanged = true;
  bool capacityChanged = true;


  // Pins that tell this MCU whether the webpage is
  // requesting to change capacity or maximum.
  pinMode(GPIOA, 0, GPIO_INPUT);// 5
  pinMode(GPIOA, 1, GPIO_INPUT);// 4
  pinMode(GPIOB, 8, GPIO_INPUT);
  pinMode(GPIOB, 9, GPIO_INPUT);

  int cupThisTime = digitalRead(GPIOA, 0);
  int cdownThisTime = digitalRead(GPIOA, 1);
  int mupThisTime = digitalRead(GPIOB, 8);
  int mdownThisTime = digitalRead(GPIOB, 9);


  int cupLastTime = cupThisTime;
  int cdownLastTime = cdownThisTime;
  int mupLastTime = mupThisTime;
  int mdownLastTime = mdownThisTime;

  while(1)
  {
    // Process IR input
    int IRResult = -1;
    if(digitalRead(GPIOA, 4) == 0)
    {
      IRResult = IRProcessing();
      delay_millis(TIM2, 100);
    }

    if(IRResult == 11101011) {togglePin(GPIOA, 1); people--; peopleChanged
= true;}
```

```
    if(IRResult == 11101110) {togglePin(GPIOA, 0); people++; peopleChanged
= true;}
    if(IRResult == 10111011) {capacity++; capacityChanged = true;}
    if(IRResult == 10111010) {capacity--; capacityChanged = true;}
    if(IRResult == 10101010) {errors = 0;}

    // Update if website requested it
    cupLastTime = cupThisTime;
    cdownLastTime = cdownThisTime;
    mupLastTime = mupThisTime;
    mdownLastTime = mdownThisTime;

    cupThisTime = digitalRead(GPIOA, 0);
    cdownThisTime = digitalRead(GPIOA, 1);
    mupThisTime = digitalRead(GPIOB, 8);
    mdownThisTime = digitalRead(GPIOB, 9);

    if (cupLastTime != cupThisTime) {people++; peopleChanged = true;}
    if (cdownLastTime != cdownThisTime) {people--; peopleChanged = true;}
    if (mupLastTime != mupThisTime) {capacity++; capacityChanged = true;}
    if (mdownLastTime != mdownThisTime) {capacity--; capacityChanged =
true;}


    // Read ADC values
    setADC(10);
    while(ADC1->ADC_SR.EOC==0){}
    int analogIn1 = ADC1->ADC_DR;

    setADC(11);
    while(ADC1->ADC_SR.EOC==0){}
    int analogIn2 = ADC1->ADC_DR;

    if (analogIn1 > Max1) {Max1 = analogIn1;}
    if (analogIn2 > Max2) {Max2 = analogIn2;}
    if (analogIn1 < Min1) {Min1 = analogIn1;}
    if (analogIn2 < Min2) {Min2 = analogIn2;}

    // If ADC values are over threshold, trigger sensor
    extSensorLastTime = extSensorThisTime;
```

```
    intSensorLastTime = intSensorThisTime;

    extSensorThisTime = analogIn1 > 1800;
    intSensorThisTime = analogIn2 > 1800;

    bool extSensorTrigger = extSensorThisTime && !extSensorLastTime;
    bool intSensorTrigger = intSensorThisTime && !intSensorLastTime;


    // FSM State
        if (state == 0 && extSensorTrigger){state = 1; timer =
99999999;}//timer = 999999;
    else if (state == 1 && intSensorTrigger){state = 0; people++;
peopleChanged = true; timer = -1; togglePin(GPIOA, 1); delay_millis(TIM2,
600);}//1
    else if (state == 0 && intSensorTrigger){state = 2; timer =
99999999;}//timer = 999999;
    else if (state == 2 && extSensorTrigger){state = 0; people--;
peopleChanged = true; timer = -1; togglePin(GPIOA, 0); delay_millis(TIM2,
600); }//0
    else if (state != 1 && timer == 0){timer = -1; state = 0; errors++;}

    if (timer > 0) {timer--;}

    // Ensure people and occupancy don't go negative
    if (people < 0) {people = 0;}
    if (capacity < 0) {capacity = 0;}

    //PA3 (2^7)
    //PA2
    //PA10
    //PB3
    //PB5
    //PB4
    //PB10
    //PA8 (least sig. bit)

    //PA9 (select)
```

```
    // Send People to FPGA (select low)
    if (peopleChanged)
    {
      digitalWrite(GPIOA, 9, 0);

      int oldPeople = people;

      if (people >= 128){people -= 128; digitalWrite(GPIOA, 3, 1);} else
{digitalWrite(GPIOA, 3, 0);}
      if (people >= 64){people -= 64; digitalWrite(GPIOA, 2, 1);} else
{digitalWrite(GPIOA, 2, 0);}
      if (people >= 32){people -= 32; digitalWrite(GPIOA, 10, 1);} else
{digitalWrite(GPIOA, 10, 0);}
      if (people >= 16){people -= 16; digitalWrite(GPIOB, 3, 1);} else
{digitalWrite(GPIOB, 3, 0);}
      if (people >= 8){people -= 8; digitalWrite(GPIOB, 5, 1);} else
{digitalWrite(GPIOB, 5, 0);}
      if (people >= 4){people -= 4; digitalWrite(GPIOB, 4, 1);} else
{digitalWrite(GPIOB, 4, 0);}
      if (people >= 2){people -= 2; digitalWrite(GPIOB, 10, 1);} else
{digitalWrite(GPIOB, 10, 0);}
      if (people >= 1){people -= 1; digitalWrite(GPIOA, 8, 1);} else
{digitalWrite(GPIOA, 8, 0);}

      people = oldPeople;

      delay_millis(TIM2, 8);
    }
    // Send Capacity to FPGA (select high)
    if (capacityChanged)
    {
      digitalWrite(GPIOA, 9, 1);

      int oldCapacity = capacity;

      if (capacity >= 128){capacity -= 128; digitalWrite(GPIOA, 3, 1);}
else {digitalWrite(GPIOA, 3, 0);}
      if (capacity >= 64){capacity -= 64; digitalWrite(GPIOA, 2, 1);} else
{digitalWrite(GPIOA, 2, 0);}
```

```
      if (capacity >= 32){capacity -= 32; digitalWrite(GPIOA, 10, 1);}
else {digitalWrite(GPIOA, 10, 0);}
      if (capacity >= 16){capacity -= 16; digitalWrite(GPIOB, 3, 1);} else
{digitalWrite(GPIOB, 3, 0);}
      if (capacity >= 8){capacity -= 8; digitalWrite(GPIOB, 5, 1);} else
{digitalWrite(GPIOB, 5, 0);}
      if (capacity >= 4){capacity -= 4; digitalWrite(GPIOB, 4, 1);} else
{digitalWrite(GPIOB, 4, 0);}
      if (capacity >= 2){capacity -= 2; digitalWrite(GPIOB, 10, 1);} else
{digitalWrite(GPIOB, 10, 0);}
      if (capacity >= 1){capacity -= 1; digitalWrite(GPIOA, 8, 1);} else
{digitalWrite(GPIOA, 8, 0);}

      capacity = oldCapacity;

      delay_millis(TIM2, 8);
    }

    peopleChanged = false;
    capacityChanged = false;
  }


}

////////////////////////////////////////////////
// Functions
////////////////////////////////////////////////

int IRProcessing()
{
  while (digitalRead(GPIOA, 4) == 0){}

  delay_micros(TIM2, 2550); //2550

  digitalWrite(GPIOB, 0, 1);

  int hold = digitalRead(GPIOA, 4);
  if (hold == 0)
  {digitalWrite(GPIOB, 0, 0);return -1;}
```

```
digitalWrite(GPIOB, 0, 0);

delay_micros(TIM2, 31100);

int waitTime = 600;

// Wait until the next bit is being displayed and read it.
// Creates a binary result
int result = 0;
digitalWrite(GPIOB, 0, 1);
result += digitalRead(GPIOA, 4);
digitalWrite(GPIOB, 0, 0);

delay_micros(TIM2, waitTime);

digitalWrite(GPIOB, 0, 1);
result += digitalRead(GPIOA, 4)*10;
digitalWrite(GPIOB, 0, 0);

delay_micros(TIM2, waitTime);

digitalWrite(GPIOB, 0, 1);
result += digitalRead(GPIOA, 4)*100;
digitalWrite(GPIOB, 0, 0);

delay_micros(TIM2, waitTime);

digitalWrite(GPIOB, 0, 1);
result += digitalRead(GPIOA, 4)*1000;
digitalWrite(GPIOB, 0, 0);

delay_micros(TIM2, waitTime);

digitalWrite(GPIOB, 0, 1);
result += digitalRead(GPIOA, 4)*10000;
digitalWrite(GPIOB, 0, 0);

delay_micros(TIM2, waitTime);
```

```
  digitalWrite(GPIOB, 0, 1);
  result += digitalRead(GPIOA, 4)*100000;
  digitalWrite(GPIOB, 0, 0);

  delay_micros(TIM2, waitTime);

  digitalWrite(GPIOB, 0, 1);
  result += digitalRead(GPIOA, 4)*1000000;
  digitalWrite(GPIOB, 0, 0);

  delay_micros(TIM2, waitTime);

  digitalWrite(GPIOB, 0, 1);
  result += digitalRead(GPIOA, 4)*10000000;
  digitalWrite(GPIOB, 0, 0);

  return result;

}

void sendString(USART_TypeDef * USART, char * str) {
  char * ptr = str; // Get a pointer to the first element in the array.

  // Check if ptr is the null terminator (i.e. 0).
  // Otherwise, send the character and post-increment the pointer to point
to
  // the next character in the string.
  while (*ptr) sendChar(USART, *ptr++);
}

int inString(char request[], char des[]) {
  if (strstr(request, des) != NULL) {return 1;}
  return -1;
}
```

ESP8266 MCU Code (runs just the ESP8266 and communicates with the Main MCU)

```
// main.c

#include "STM32F401RE_FLASH.h"
#include "STM32F401RE_RCC.h"
```

```
#include "STM32F401RE_USART.h"
#include "STM32F401RE_GPIO.h"
#include "STM32F401RE_SPI.h"
#include <string.h> // for strstr()
#include <stdint.h> // for integer types (i.e., uint32_t)
#include <stdio.h>  // for sprintf()


#define USART_ID USART1_ID

#define BUFF_LEN 32

/////////////////////////////////////////////////////////////////
// Provided Constants and Functions
/////////////////////////////////////////////////////////////////

//Defining the web page in two chunks: everything before the current time,
and everything after the current time
//Please see the e155 website for a human-readable version of the file
"webpage.html"
char * webpageStart = "<!DOCTYPE html><html><head><title>Final Project
Demo</title><meta name=\"viewport\" content=\"width=device-width,
initial-scale=1.0\"></head><body><h1>Marz and Max Covid Police Capacity
Monitoring System</h1>";
char * currentCapstr = "<form action=\"cup\"><input type=\"submit\"
value=\"Increase current capacity\" /></form>\
  <form action=\"cdown\"><input type=\"submit\" value=\"Decrease current
capacity\" /></form>";
char * maxCapstr = "<form action=\"mup\"><input type=\"submit\"
value=\"Increase maximum capacity\" /></form>\
<form action=\"mdown\"><input type=\"submit\" value=\"Decrease maximum
capacity\" /></form>";
char * webpageEnd   = "</body></html>";

// Sends a null terminated string of arbitrary length
void sendString(USART_TypeDef * USART, char * str) {
  char * ptr = str; // Get a pointer to the first element in the array.

  // Check if ptr is the null terminator (i.e. 0).
```

```
  // Otherwise, send the character and post-increment the pointer to point
to
  // the next character in the string.
  while (*ptr) sendChar(USART, *ptr++);
}

//determines whether a given character sequence is in a char array
request, returning 1 if present, -1 if not present
int inString(char request[], char des[]) {
  if (strstr(request, des) != NULL) {return 1;}
  return -1;
}

uint8_t updateCurrentCap(char request[], uint8_t currCapacity) {
  // The request has been received. now process to determine whether to
turn the LED on or off
  if (inString(request, "cup") == 1) {
    togglePin(GPIOA, 5);
    currCapacity += 1;
  }

  if (inString(request, "cdown") == 1) {
    togglePin(GPIOA, 4);
    currCapacity -= 1;
  }

  return currCapacity;
}

uint8_t updateMaximumCap(char request[], uint8_t maxCapacity) {
  // The request has been received. now process to determine whether to
turn the LED on or off
  if (inString(request, "mup") == 1) {
    togglePin(GPIOB, 8);
    maxCapacity += 1;
  }

  if (inString(request, "mdown") == 1) {
    togglePin(GPIOB, 9);
    maxCapacity -= 1;
```

```
  }

  return maxCapacity;
}

/////////////////////////////////////////////////////////////////
// Other Functions
/////////////////////////////////////////////////////////////////

void sendHTML(USART_TypeDef * ESPUSART, char maxCapacity[128], char
currCapacity[128], char overCapacity[128]){
    // Transmit the webpage over UART by sending a series of strings:
    sendString(ESPUSART, webpageStart);
    sendString(ESPUSART, "<br>"); // Line break
    sendString(ESPUSART, "<h2>Maximum Capacity:</h2>");
    sendString(ESPUSART, maxCapacity);
    sendString(ESPUSART, "<br><br>"); // Line break
    sendString(ESPUSART, "Update Maximum Capacity: ");
    sendString(ESPUSART, maxCapstr);
    sendString(ESPUSART, "<br>"); // Line break
    sendString(ESPUSART, "<h2>Current Capacity:</h2>");
    sendString(ESPUSART, currCapacity);
    sendString(ESPUSART, "<br><br>"); // Line break
    sendString(ESPUSART, "Update Current Capacity: ");
    sendString(ESPUSART, currentCapstr);
    sendString(ESPUSART, "<br>"); // Line break
    sendString(ESPUSART, "<h2>");
    sendString(ESPUSART, overCapacity);
    sendString(ESPUSART, "</h2>");
    sendString(ESPUSART, webpageEnd);
}

uint8_t readCapacity(){
    uint8_t capacity = 0;
    if(digitalRead(GPIOA, 7)==1)  capacity = 64;
    if(digitalRead(GPIOA, 0)==1) capacity += 32;
    if(digitalRead(GPIOB, 3)==1)  capacity += 16;
    if(digitalRead(GPIOB, 5)==1)  capacity += 8;
    if(digitalRead(GPIOB, 4)==1)  capacity += 4;
    if(digitalRead(GPIOB, 10)==1)  capacity += 2;
```

```
    if(digitalRead(GPIOA, 8)==1)  capacity += 1;
    return capacity;
}

/////////////////////////////////////////////////////////////////
// main()
/////////////////////////////////////////////////////////////////

int main(void) {

    // Configure flash and clock
    configureFlash();
    configureClock(); // Set system clock to 84 MHz

    // Turn on GPIOA and GPIB
    RCC->AHB1ENR.GPIOAEN = 1;
    RCC->AHB1ENR.GPIOBEN = 1;


    // LED Matrix Pins
    //PA7 (2^6)      changed from PA2
    //PA0           changed from PA10
    //PB3
    //PB5
    //PB4
    //PB10
    //PA8 (least)

    //PA1 (select)  changed from PA9

    // To read capacity:
    pinMode(GPIOA, 7, GPIO_OUTPUT); // changed from PA2
    pinMode(GPIOA, 0, GPIO_OUTPUT); // changed from PA10
    pinMode(GPIOB, 3, GPIO_OUTPUT);
    pinMode(GPIOB, 5, GPIO_OUTPUT);
    pinMode(GPIOB, 4, GPIO_OUTPUT);
    pinMode(GPIOB, 10, GPIO_OUTPUT);
    pinMode(GPIOA, 8, GPIO_OUTPUT);

    pinMode(GPIOA, 1, GPIO_OUTPUT); // changed from PA9
```

```
    // To update capacity:
    //PA5: increase current
    //PB5: decrease current
    //PB4: increase max
    //PB10: decrease max
    pinMode(GPIOA, 5, GPIO_OUTPUT); // plugged into PA5
    pinMode(GPIOA, 4, GPIO_OUTPUT); // plugged into PA4
    pinMode(GPIOB, 8, GPIO_OUTPUT); // plugged into PA8
    pinMode(GPIOB, 9, GPIO_OUTPUT); // plugged into PA9

    // Initialize the UART connection for the ESP8266
    USART_TypeDef * ESPUSART = initUSART(USART1_ID);  // USART using PA9
and PA10

    uint8_t capacity, maxCapacity, currCapacity = 0;

    while(1) {
    /* Wait for ESP8266 to send a request.
    Requests take the form of '/REQ:<tag>\n', with TAG being <= 10
characters.
    Therefore the request[] array must be able to contain 18 characters.
    */

      // Receive web request from the ESP
      char request[BUFF_LEN] = "                  "; // initialize to
known value
      int charIndex = 0;

      // Keep going until you get end of line character
      while(inString(request, "\n") == -1) {
        // Wait for a character to be received before reading the RX
buffer
        while(!ESPUSART->SR.RXNE){
            //capacity = readCapacity();
            //if(digitalRead(GPIOA, 1)==1) maxCapacity = capacity;
            //else currCapacity = capacity;

        }
        request[charIndex++] = (char) receiveChar(ESPUSART);
```

```
      }

      capacity = readCapacity();
      if(digitalRead(GPIOA, 1)==1) maxCapacity = capacity;
      else currCapacity = capacity;

      currCapacity = updateCurrentCap(request, currCapacity);
      maxCapacity = updateMaximumCap(request, maxCapacity);

      for(int i = 0; i < 99; i++);

      char maxCap[128], currCap[128], overCap[128];
      sprintf(maxCap, "The Maximum Capacity is %d!", maxCapacity);
      sprintf(currCap, "The Current Capacity is %d!", currCapacity);

      if(maxCapacity<=currCapacity) sprintf(overCap, "We are currently
over capacity!");
      else sprintf(overCap, "We are currently under capacity, free to
enter :)");


      // Send HTML webpage to ESP
      sendHTML(ESPUSART, maxCap, currCap, overCap);

    }

}
```

Verilog Code

```
/*
  LED Matrix Display
  (Marz Barnes 11-23-2021)

  Pin    ments
      PIN_H6:  clk_in, 12MHz clock
      PIN_H10: capacitySelect (1: max capacity, 0: current capacity)

      PIN_H5:  blank
      PIN_G12: ltch (latch)
      PIN_H13: sclk

      PIN_D1:  A0
      PIN_E3:  A1
      PIN_E4:  A2
      PIN_H8:  A3

      PIN_H4:  R0
      PIN_J1:  G0
      PIN_J2:  B0
      PIN_C2:  R1
      PIN_F1:  G1
      PIN_C1:  B1

      PIN_K10: cap6
      PIN_L12: cap5
      PIN_J12: cap4
      PIN_J13: cap3
      PIN_K11: cap2
      PIN_K12: cap1
      PIN_J10: cap0
*/

```

// Top Level Module: FPGA Driver

```
////////////////////////////////////////////////////////////////////////
// FPGA_driver
// top level module
//
////////////////////////////////////////////////////////////////////////
module FPGA_driver(input logic clk, capacitySelect,
                   input logic cap0, cap1, cap2, cap3, cap4, cap5, cap6,
                   output logic blank, ltch,
                   output logic R0, G0, B0, R1, G1, B1,
                   output logic A0, A1, A2, A3,
                   output logic sclk);

    logic clk_out;
    logic [5:0] col = 0;
    logic [3:0] A;
    logic [3:0] nextA = 0;
    logic [31:0] Rstring_0, Gstring_0, Bstring_0, Rstring_1, Gstring_1, Bstring_1;
    logic [7:0] capacity;
    logic [3:0] currentRow;
    logic [2:0] RGB0;
    logic [2:0] RGB1;

    clk_gen slow(clk, 0, clk_out);

    // row, col state register
    always_ff @(posedge clk_out) begin
        if(col==0) begin
            currentRow = nextA+3; // add 3 to shift the display to the top of the matrix
            nextA = nextA + 1;
        end
        col = col + 1;
    end

    // Controller for creating display from maximum and current capacity inputs:
    //
    assign capacity = {cap6, cap5, cap4, cap3, cap2, cap1, cap0};
    set_next_RGBstring setNext(clk_out, capacitySelect, capacity, col, currentRow,
                               Rstring_0, Gstring_0, Bstring_0,
                               Rstring_1, Gstring_1, Bstring_1);


    // Controller for displaying on the LED matrix:
    //
    matrix_row_FSM flashRow(clk_out, 0,
                            Rstring_0, Gstring_0, Bstring_0,
                            Rstring_1, Gstring_1, Bstring_1,
                            col, nextA, A, RGB0, RGB1, blank, ltch, sclk);

    //Output Logic:
    always_comb begin
        {R0, G0, B0} = RGB0;
        {R1, G1, B1} = RGB1;
        {A3, A2, A1, A0} = A;
        //sclk = clk_out;
    end

endmodule

```

// Matrix Display Logic: set_next_RGBstring

```
//////////////////////////////////////////////////////////////////////
// set_next_RGBstring
// stores matrix pattern and updates based on capacity data
// returns next RGB for current row being displayed
//
//////////////////////////////////////////////////////////////////////
module set_next_RGBstring(input logic clk, capacitySelect,
                          input logic [6:0] capacity,
                          input logic [5:0] col,
                          input logic [3:0] currentRow,
                          output logic [31:0] Rstring_0, Gstring_0, Bstring_0,
                          output logic [31:0] Rstring_1, Gstring_1, Bstring_1);

   logic [3:0] maxcapDigit2, maxcapDigit1, maxcapDigit0, currcapDigit2, currcapDigit1, currcapDigit0;
   logic [6:0] maxcap, currcap;
   logic [31:0] topdigitMatrix [5];
   logic [31:0] botdigitMatrix [5];
   logic [31:0] topMatrix [16];
   logic [31:0] botMatrix [16];

   // Update maxcapacity and current capacity from asynchronous logic inputs:
   // (updates only at the start of a display cycle)
   always_ff @(posedge clk) begin
      if(currentRow==0 && col==0) begin
         if(capacitySelect==1) maxcap = capacity;
         else currcap = capacity;
      end
   end

   // Combinational logic to update display matrix:

   find_Digits digitMatrix1(maxcap, topdigitMatrix);
   find_Digits digitMatrix2(currcap, botdigitMatrix);

   always_comb begin
      topMatrix[0]  = 32'b00000000000000000000000000000000;

      topMatrix[1]  = 32'b00000001000100001000010001000000;
      topMatrix[2]  = 32'b00000001101100010100001010000000;
      topMatrix[3]  = 32'b00000001010100100010000100000000;
      topMatrix[4]  = 32'b00000001000100111110001010000000;
      topMatrix[5]  = 32'b00000001000100100010010001000000;

      topMatrix[6]  = 32'b01110010011100100111011101110101;
      topMatrix[7]  = 32'b01000101010101010100001000100101;
      topMatrix[8]  = 32'b01000111011101110100001000100010;
      topMatrix[9]  = 32'b01110101010001010111011100100010;
      topMatrix[10] = 32'b00000000000000000000000000000000;

      topMatrix[11] = topdigitMatrix[0];
      topMatrix[12] = topdigitMatrix[1];
      topMatrix[13] = topdigitMatrix[2];
      topMatrix[14] = topdigitMatrix[3];
      topMatrix[15] = topdigitMatrix[4];

      botMatrix[0]  = 32'b00000000000000000000000000000000;

      botMatrix[1]  = 32'b00000001111100111110011111000000;
      botMatrix[2]  = 32'b00000001000000100000000100000000;
      botMatrix[3]  = 32'b00000001111100111110000100000000;
      botMatrix[4]  = 32'b00000001000000000010000100000000;
      botMatrix[5]  = 32'b00000001111100111110000100010000;

      botMatrix[6]  = 32'b01110010011100100111011101110101;
      botMatrix[7]  = 32'b01000101010101010100001000100101;
      botMatrix[8]  = 32'b01000111011101110100001000100010;
      botMatrix[9]  = 32'b01110101010001010111011100100010;

      botMatrix[10] = 32'b00000000000000000000000000000000;

      botMatrix[11] = botdigitMatrix[0];
      botMatrix[12] = botdigitMatrix[1];
      botMatrix[13] = botdigitMatrix[2];
      botMatrix[14] = botdigitMatrix[3];
      botMatrix[15] = botdigitMatrix[4];
   end
```

```
   // Output Register
   always_ff @(posedge clk) begin
      if (col==0) begin
         if(maxcap <= currcap) begin          // Flash red when over or at max capacity
            Rstring_0 = topMatrix[currentRow];
            Rstring_1 = botMatrix[currentRow];
            Gstring_0 = 0;
            Gstring_1 = 0;
         end
         else begin                           // Flash white when under max capacity
            Rstring_0 = topMatrix[currentRow];
            Rstring_1 = botMatrix[currentRow];
            Gstring_0 = topMatrix[currentRow];
            Gstring_1 = botMatrix[currentRow];
         end

         if(currentRow >= 11 && currentRow <=15) begin
            Bstring_0 = topMatrix[currentRow];
            Bstring_1 = botMatrix[currentRow];
         end
         else begin
            Bstring_0 = 0;
            Bstring_1 = 0;
         end

      end
   end

endmodule
```

// Setting registers in LED Display: matrix_row_FSM

```
////////////////////////////////////////////////////////////////////////
// matrix_row_FSM
// displays single row on LED matrix from string of column data
//
////////////////////////////////////////////////////////////////////////
module matrix_row_FSM(input logic clk, start,
                      input logic [31:0] Rstring_0, Gstring_0, Bstring_0,
                      input logic [31:0] Rstring_1, Gstring_1, Bstring_1,
                      input logic [5:0] col,
                      input logic [3:0] nextA,
                      output logic [3:0] A,
                      output logic [2:0] RGB0, RGB1,
                      output logic blank, ltch, sclk);

   //logic [3:0] nextA = 0;
   //logic [5:0] col = 0;
   logic [2:0] state = 0;
   logic [2:0] nextState = 0;

   // State Register
   always_ff @(posedge clk) begin
      if(start) begin
         // nextA = 0;
         // col = 0;
         state = 0;
      end

      //if(col==0) nextA = nextA + 1;
      state = nextState;
      //col = col + 1;
   end


   // Next State Logic
   always_comb begin
      case(state)
         3'b000:  if(col==32) nextState = 3'b001;  // S0: Shift pixel data
                  else        nextState = 3'b000;
         3'b001:  if(col==33) nextState = 3'b010;  // S1: Assert blank
                  else        nextState = 3'b001;
         3'b010:  if(col==36) nextState = 3'b011;  // S2: Latch
                  else        nextState = 3'b010;
         3'b011:  if(col==37) nextState = 3'b100;  // S3: Set Address
                  else        nextState = 3'b011;
         3'b100:  if(col==63) nextState = 3'b000;  // S4: Blank display
                  else        nextState = 3'b100;
         default:             nextState = 3'b000;
      endcase
   end
```

```
	// Next State Logic
	always_comb begin
		case(state)
			3'b000:	if(col==32) nextState = 3'b001;	// S0: Shift pixel data
				else		nextState = 3'b000;
			3'b001:	if(col==33) nextState = 3'b010;	// S1: Assert blank
				else		nextState = 3'b001;
			3'b010:	if(col==36) nextState = 3'b011;	// S2: Latch
				else		nextState = 3'b010;
			3'b011:	if(col==37) nextState = 3'b100;	// S3: Set Address
				else		nextState = 3'b011;
			3'b100:	if(col==63) nextState = 3'b000;	// S4: Blank display
				else		nextState = 3'b100;
			default:		nextState = 3'b000;
		endcase
	end


	// Send out clk only while RGB data being transmitted
	always_comb begin
		if(state==0)	sclk = clk;
		else		sclk = 0;
	end

	// Output Logic [RGB, sclk]
	always_ff @(negedge clk) begin
		if(state==0)	begin RGB0	= {Rstring_0[31-col], Gstring_0[31-col], Bstring_0[31-col]};
				      RGB1	= {Rstring_1[31-col], Gstring_1[31-col], Bstring_1[31-col]};
				end
		else		begin RGB0 = 3'b000;
				      RGB1 = 3'b000;
				end
	end


	// Output Logic [blank, latch, address]
	always_ff @(negedge clk) begin
		if(state==0)		begin blank = 0;
					      ltch  = 0;
					      A     = A;	// S0: Shift in Pixel Data
					end
		else if(state==1) begin blank = 1;
					      ltch  = 0;
					      A     = A;	// S1: Assert Blank
					end
		else if(state==2) begin blank = 1;
					      ltch  = 1;
					      A     = A;	// S2: Latch
					end
		else if(state==3) begin blank = 1;
					      ltch  = 0;
					      A     = nextA; // S3: Set Address
					end
		else if(state==4) begin blank = 0;
					      ltch  = 0;
					      A     = nextA; // S4: Blank display
					end
		else			begin blank = 0;
					      ltch  = 0;
					      A     = A;
					end
	end
endmodule
```

// clk_gen: outputs slow clk

```

////////////////////////////////////////////////////////////////////////
// clk_gen
// generating a clk at the appropriate frequency
//
////////////////////////////////////////////////////////////////////////
module clk_gen(input logic clk_in, clk_reset,
		output logic clk_out);

	logic [6:0] counter = 0;

	always_ff @(posedge clk_in) begin

		if (clk_reset == 1'b1) begin
			clk_out = 0;
			counter = 0;
		end

		counter = counter + 1;
		clk_out = counter[6];

	end

endmodule
```

// find_Digits: determines 5x32 matrix from capacity input

```
//////////////////////////////////////////////////////////////////////
// find_Digits
// updates bottom row of display with capacity value
// (combinational logic)
//
//////////////////////////////////////////////////////////////////////
module find_Digits(input logic [7:0] capacity,
                   output logic [31:0] digitMatrix [5]);

   logic [4:0] digitMatrix2 [5];
   logic [4:0] digitMatrix1 [5];
   logic [4:0] digitMatrix0 [5];
   logic [3:0] digit2, digit1, digit0;

   always_comb begin
      digit2 = capacity/100;
      digit1 = (capacity%100)/10;
      digit0 = capacity%10;
   end

   digit findDigit2(digit2, digitMatrix2);
   digit findDigit1(digit1, digitMatrix1);
   digit findDigit0(digit0, digitMatrix0);

   always_comb begin
      digitMatrix[0]  = {7'b0000000, digitMatrix2[0], 2'b00, digitMatrix1[0], 2'b00, digitMatrix0[0], 6'b000000};
      digitMatrix[1]  = {7'b0000000, digitMatrix2[1], 2'b00, digitMatrix1[1], 2'b00, digitMatrix0[1], 6'b000000};
      digitMatrix[2]  = {7'b0000000, digitMatrix2[2], 2'b00, digitMatrix1[2], 2'b00, digitMatrix0[2], 6'b000000};
      digitMatrix[3]  = {7'b0000000, digitMatrix2[3], 2'b00, digitMatrix1[3], 2'b00, digitMatrix0[3], 6'b000000};
      digitMatrix[4]  = {7'b0000000, digitMatrix2[4], 2'b00, digitMatrix1[4], 2'b00, digitMatrix0[4], 6'b000000};
   end

endmodule
```

// digits: determines 5x5 matrix from digit input

```
//////////////////////////////////////////////////////////////////////
// digit
// returns a single 5x5 digit matrix from desired digit to be displayed
// (combinational logic)
//
//////////////////////////////////////////////////////////////////////
module digit(input logic [3:0] digit,
             output logic [4:0] digitMatrix [5]);

   always_comb begin
      case(digit)
         4'b0000:    begin digitMatrix[0] = 5'b11111;
                           digitMatrix[1] = 5'b10001;
                           digitMatrix[2] = 5'b10001;
                           digitMatrix[3] = 5'b10001;
                           digitMatrix[4] = 5'b11111;
                     end
         4'b0001:    begin digitMatrix[0] = 5'b00100;
                           digitMatrix[1] = 5'b01100;
                           digitMatrix[2] = 5'b10100;
                           digitMatrix[3] = 5'b00100;
                           digitMatrix[4] = 5'b11111;
                     end
         4'b0010:    begin digitMatrix[0] = 5'b11111;
                           digitMatrix[1] = 5'b00001;
                           digitMatrix[2] = 5'b11111;
                           digitMatrix[3] = 5'b10000;
                           digitMatrix[4] = 5'b11111;
                     end
         4'b0011:    begin digitMatrix[0] = 5'b11111;
                           digitMatrix[1] = 5'b00001;
                           digitMatrix[2] = 5'b11111;
                           digitMatrix[3] = 5'b00001;
                           digitMatrix[4] = 5'b11111;
                     end
         4'b0100:    begin digitMatrix[0] = 5'b10001;
                           digitMatrix[1] = 5'b10001;
                           digitMatrix[2] = 5'b11111;
                           digitMatrix[3] = 5'b00001;
                           digitMatrix[4] = 5'b00001;
                     end
         4'b0101:    begin digitMatrix[0] = 5'b11111;
                           digitMatrix[1] = 5'b10000;
                           digitMatrix[2] = 5'b11111;
                           digitMatrix[3] = 5'b00001;
                           digitMatrix[4] = 5'b11111;
                     end
         4'b0110:    begin digitMatrix[0] = 5'b11111;
                           digitMatrix[1] = 5'b10000;
                           digitMatrix[2] = 5'b11111;
                           digitMatrix[3] = 5'b10001;
                           digitMatrix[4] = 5'b11111;
                     end
```

```
            4'b0101:    begin digitMatrix[0] = 5'b11111;
                              digitMatrix[1] = 5'b10000;
                              digitMatrix[2] = 5'b11111;
                              digitMatrix[3] = 5'b00001;
                              digitMatrix[4] = 5'b11111;
                        end
            4'b0110:    begin digitMatrix[0] = 5'b11111;
                              digitMatrix[1] = 5'b10000;
                              digitMatrix[2] = 5'b11111;
                              digitMatrix[3] = 5'b10001;
                              digitMatrix[4] = 5'b11111;
                        end
            4'b0111:    begin digitMatrix[0] = 5'b11111;
                              digitMatrix[1] = 5'b00001;
                              digitMatrix[2] = 5'b00001;
                              digitMatrix[3] = 5'b00001;
                              digitMatrix[4] = 5'b00001;
                        end
            4'b1000:    begin digitMatrix[0] = 5'b11111;
                              digitMatrix[1] = 5'b10001;
                              digitMatrix[2] = 5'b11111;
                              digitMatrix[3] = 5'b10001;
                              digitMatrix[4] = 5'b11111;
                        end
            4'b1001:    begin digitMatrix[0] = 5'b11111;
                              digitMatrix[1] = 5'b10001;
                              digitMatrix[2] = 5'b11111;
                              digitMatrix[3] = 5'b00001;
                              digitMatrix[4] = 5'b11111;
                        end
            default:    begin digitMatrix[0] = 5'b00001;
                              digitMatrix[1] = 5'b01000;
                              digitMatrix[2] = 5'b00000;
                              digitMatrix[3] = 5'b00001;
                              digitMatrix[4] = 5'b01000;
                        end
        endcase
    end
endmodule
```