Wilson Au-Yeung

CMPE 121 L

Anujan Varma

November 13, 2016

Lab Report 5

Introduction:

This lab was comprised of two sections, each of which were focused on teaching us the basics of using and getting familiar with the LED display using row & column multiplexing. We first had to read the description on how the RGB LED panel worked and what inputs were needed in order to display the right LEDS. Each LED were separated into 16 rows and 32 columns, but in this particular lab, we needed to only turn on and control a 4x4 RGB matrix display. The methods used to display the LEDs for part 1 and part 2 were based on an interrupt driven refresh and a hardware controlled refresh. The first part of the lab comprised of hard coding the steps needed to turn on each LED and additional outputs which helped make it happen such as the latch and the OE. The second part of the lab was implementing a similar design by making a hardware controller out of the UDB blocks.

Set Up:

Part 0: Setting up the board

Before implementing the software implementation of the LED display, we have to set up the LED board matching up I/O Outputs to its appropriate pins on the PSoc Controller. We followed the documentation and the picture on the Lab Manual. The instructions on how to turn on a row of LEDs consisted of controlling the outputs of the rows: A, B, C , the output enable and the latch. So this provided a clear guide on every output needed to send to the LED display in order to turn on and control the RGB LED display. We used the Ribbon cable and the provided wires to connect the pins to the back input panel.

The instructions told us to implement a test design of what we were supposed to code turning on the first 2 rows of LEDs.

1) This included serial inputs such as R, G, and B that determined which of the 3 colors would turn on. We were told to set Red to high and the rest to low.
2) The next step was to generate 32 pulses on the CLK input by hard setting the output of CLK to high and low. This determined which column to write the color as it went through the corresponding Row through inputs a, b, and c.
3) The next step was to generate a single pulse on the LAT input which would hold the pattern of the LED. Then the pattern should display on the board.
4) Then we were told to set the OE input to low so it would enable the display.
5) Lastly, we needed to implement an ISR to refresh the board in order to eliminate the flicker. I did this by implementing an ISR interrupt from a timer every time it hit 1ms so it would refresh at a high frequency.

Part 1: Interrupt-Driven Refresh

Since we were able to set up part 0, all the wires are connected properly to its respective GPIO Pins. And because we familiarized ourselves with the test code, we know how each individual output pin affects the RGB LED panel. On the top design schematic, we used control registers to hold its values for its corresponding LEDs. If we write 100 to it, the LED would be set red. If we write 010 to it, the LED would be set green. And lastly 001 would be blue. The top design also contained a timer that was used for the time interrupts every millisecond so it prevents ghosting / flickering on the LED panel. The ISR would occur every time the timer reaches its terminal count which would be 1 millisecond. I established an if statement inside the main function which only allowed the main code to be executed if the flag was turned on from the ISR. (Figure 1)

```
if(flag == 1){
```

```
CY_ISR(ISR)
{
    if(flag == 0){
        flag = 1;
    }
    Timer_ReadStatusRegister();
}
```

Figure 1: Implementation of Interrupt Refresh Using Timer ISR

By implementing this refresh, this allows the LEDs to turn on one row per millisecond which is fast enough to be unseen to the eye.

Part 2: Hardware Controller

This part of the lab requires us to design a hardware controller for the LED panel using UDB blocks to implement the same idea as in part 1. We were told to use 16 control registers (each with 3 bits) so we could retain and input values as needed for each of the 3 LEDs (RGB) in each panel. I separated the control registers in the way they would be displayed on the 4x4 LED board. (Figure 2)

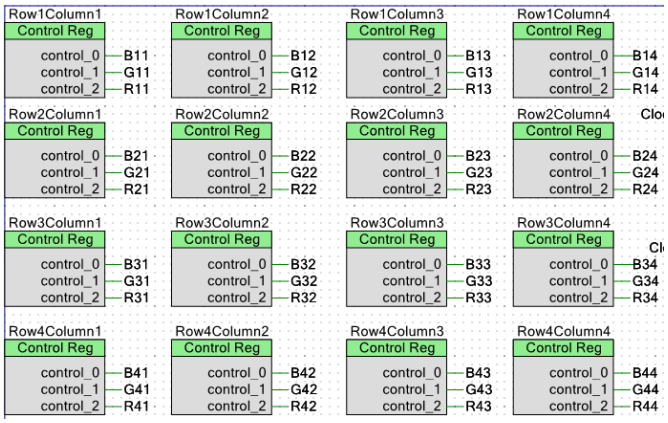

Figure 2: Control Registers Representing LEDs

The hardware controller that we would be building would take in the inputs from the control registers and inserted into a 16 input mux with a 4 bit selector. We would have 3 different muxes to keep track of each different color: RGB. The 4 bit selector would consist of 2 bits containing the row of the specified control register, and 2 bits of the specified column registers. Each time the clock cycles, it would enable the columns associated with the row on the LED.

Part 3: PWM Control of LED Brightness

This part of the Lab required us to control and vary the LED brightness of part 2. We were told to modify part two by adding a PWM which would allow us to control the relative brightness of the LEDs. We were also told to design it to allow 256 levels of brightness. Looking back, all we had to do was to implement the PWM and ADC from lab 1 and add it to our top design. We had the ADC to convert the analog signal to digital to write values to change the compare value for the PWM. I used the potentiometer on the board so I did not have to add more wires onto my Psoc Controller.

Procedure:

Part 1: Interrupt-Driven Refresh

The implementation of the software for the Interrupt-Driven Refresh was a bit tricky. First we would turn on the OE. Then we would write the value of the row inside the control register that would output to output pins ABC. Then I implemented a for loop allowing the clock pin to pulse 32, once for each column. I created a masked initialized at zero. I checked each individual array checking its values for high in each of the three colors. If any of them were on, I Logic ORed them in order to display multiple colors. Then I wrote the value of the color/mask (Figure 3) inside of the control register.

```
for(column = 0; column < 32 ; column++){
    mask = 0;
    if(column < 4){
        if(red[row][column] == 1){
            mask = mask | 0b100;
        }
        if(green[row][column] == 1){
            mask = mask | 0b010;
        }
        if(blue[row][column] == 1){
            mask = mask | 0b001;
        }
    }
```

Figure 3: Implementing Masks to Turn on LEDs

Lastly I would pulse the clock once representing one column of the 32 per row. Then I would pulse latch once and set OE to low. Setting the flag to zero allows it to only run once through every time the ISR goes off. The only code written in the ISR is turning the flag on. Incrementing the row variable would just write into a new value for the row control register each time it runs. It then checks it till it reaches 4 and resets to 0 so it would not go past that.

Part 2: Hardware Controller

This part was done all in the schematic. There was absolutely no code in the main function. To implement the hardware controller, I had to implement 2 different counters. One counter for the columns which went up to 32. And another counter which went up to 4 which counted the rows. I connected a clock into the column counter, and used a comparator to send a high once it hits

32. That makes it so it would iterate through the columns and would increment the rows once it's done with one column. (Figure 4)

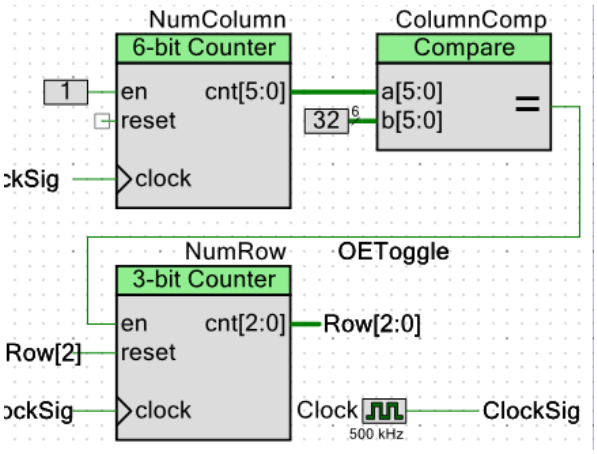

Figure 4: Counters to keep track of Columns and Rows

I connected the enable of the comparator and inverted it and outputted to the OE. I did that because it would stay high and off once after it is done with one row like how it did in the software implementation. And the Latch would only turn on for a moment immediately after. As said before, the first two selectors for my mux were my selectors for rows and the last two were the column selectors. (Figure 5) In order to turn these 2 separate 2 bit buses into a 4 bit bus for the actual selector, I inputted them into a lookup table. I was able to use the LUT as a buffer which allowed me to output whatever I inputted into it. (Figure 6)

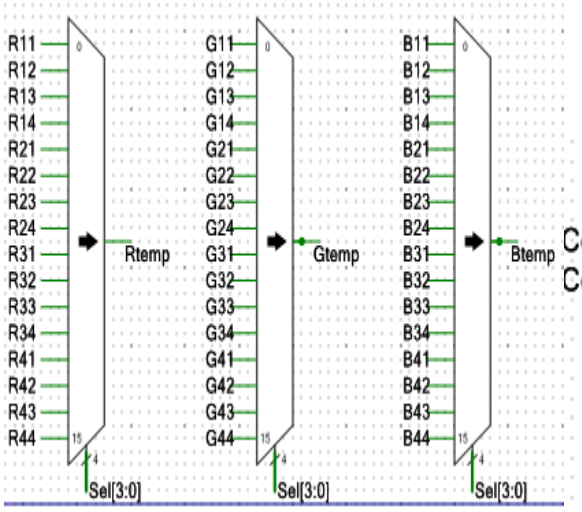

Figure 5: Mux Implementation

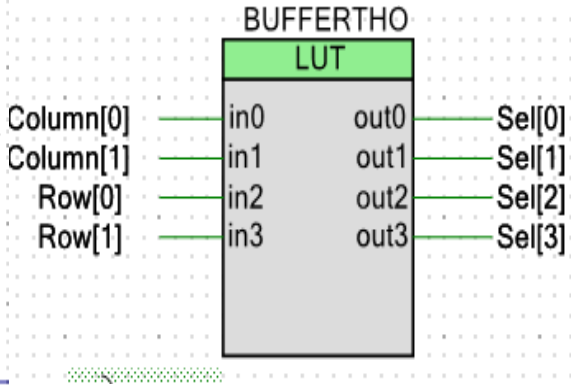

Figure 6: Look-Up Table Buffer

Part 3: PWM Control of LED Brightness

This implementation was very easy due to the fact that we were allowed to keep our schematic of Part two and use Lab one resources to implement it. I added the ADC to convert the analog value of my potentiometer to digital. Then I wrote that as the compare value inside the PWM. I used the PWM as an input for a 2 input mux. The other input was a logic zero. Selector was the output for the R, G, or B 16 input mux. This allowed the PWM to only go through if the LED was supposed to be turned on during that time. (Figure 7)

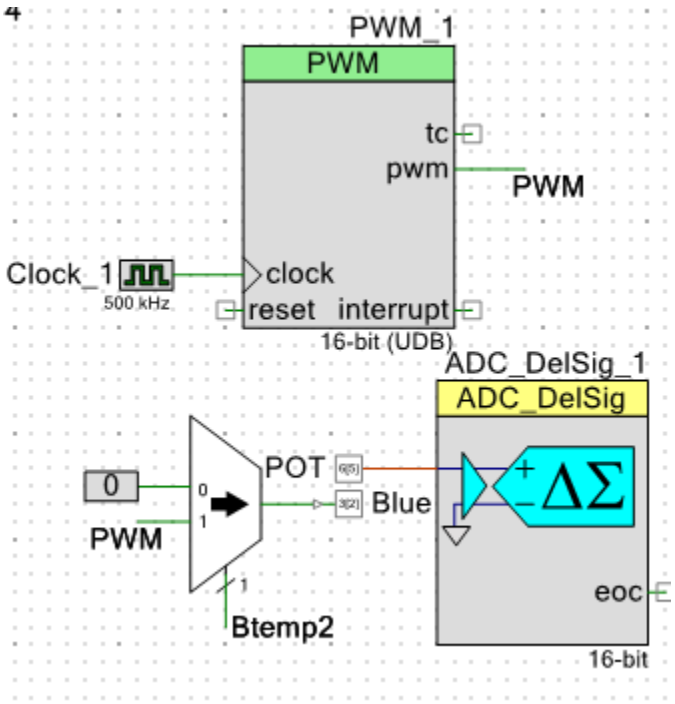

Figure 7: PWM implementation

Conclusions:

1) Discuss the advantages and disadvantages of the two methods. Which one will you use for your final project if you have to extend the design to control the entire 16x32 RGB array?

Interrupt-Driven Refresh

Pros: It allows us to directly control Inputs and Outputs in which particular order. (More Control) Direct input into the control registers was easy, but writing values into the arrays. Easier to debug with less things happening simultaneously like the Top Design in the Hardware Control

Cons: Would be very complex if more interrupts were happening at the same time due to priority interrupts that has to wait to be serviced. Harder to implement Logic statements (have to add masks in order to do the same task of adding or gates). Setting times for the ISR might be too short of a cycle especially to run through all the LEDs on the board rather than just 16.

Hardware Controller

Pros: Very Logical, Speed, Each Control Register represents a LED (Visually Better), Can actually see where different outputs are going to and where they are coming from

Cons: Impractical for Larger quantities of LEDs. Harder to control inputs and outputs at the time you want. There is no software integration using only UDBs. Gets unmanageably messy quick because of the different components to keep track of.

I would personally use the interrupt driven refresh method over hardware controlled because it would be easier to visualize by knowing exactly what inputs are being inputted at the time. So it would be easier to manage and debug. It probably won't be as fast because there is a refresh every 1ms but it wont get as messy as hardware control when I have to keep track of all those LEDS.

2) One of the limitations of these designs is that all the LEDs have the same duty cycle (1/8). This limits the number of shades available to 8, and you will not be able to get some of the colors in the image on Page 1. How will you enhance your design in Part 2 to support 4096 different colors?

I would write different compare values / change duty cycle into the PWM just like I did in part 3. The only thing different would be to not always connect the same pulse to all three different colors. Or that would only dim the LEDS with the same color as before.