For my custom lab project, I recreated the Simon game with a breadboard that contained four LEDs, four buttons, a microcontroller, a LCD screen and a speaker which are all connected to power and ground. WIthin the program, I used multiple libraries to assist in creating the program. The io.c and io.h files are there to help access the LCD as well as open us to use certain functions that connect with the LCD display. I included the <stdlib.h> file for the random number generator that I used with the srand() function as well as a random function. I tested both, but since the program is not connected with the computer clock. The random numbers that are generated are repeated if the player attempts to start a new game. I also include a <util/delay.h> file for the timing of the output and inputs. Since we had already used a timer that sets flags and interrupt, I was wondering if it was possible to replace. Although the timer file and functions are more efficient, I was able to implement a new file that could travel efficiently through the code and still give me my desired results. I set two separate arrays for the game. One array was for the numbers that were randomly given from the generator. The other array is for storing the inputs after each sequence. Old inputs from before will constantly overwrite to ensure that the player consistently gets it right after just one time.

Since the Simon game is a memory game based on flashing LEDs, I have it so that I set a random number generator function to output a certain LED as well as a certain sound starting with one.The LCD will keep track of the player's score and provide instructions when the game is over or still occurring. After the LED is outputted, the player is required to press a button which will output a certain LED and if that button pressed is the same as the LED outputted, the sequence will increment and keep going until 9 where the player will achieve victory. The rules for this game is very simple. Follow the patterns of the LED outputted and press the buttons that correspond with the LEDs and noises in that same pattern. The more the player gets right, the harder the game gets. The sound and LED outputs should correspond to the inputs the player puts in. Once an incorrect input is pressed, the game will automatically stop and reset bringing the player back to the welcome screen and waits for any input in order to restart the game.

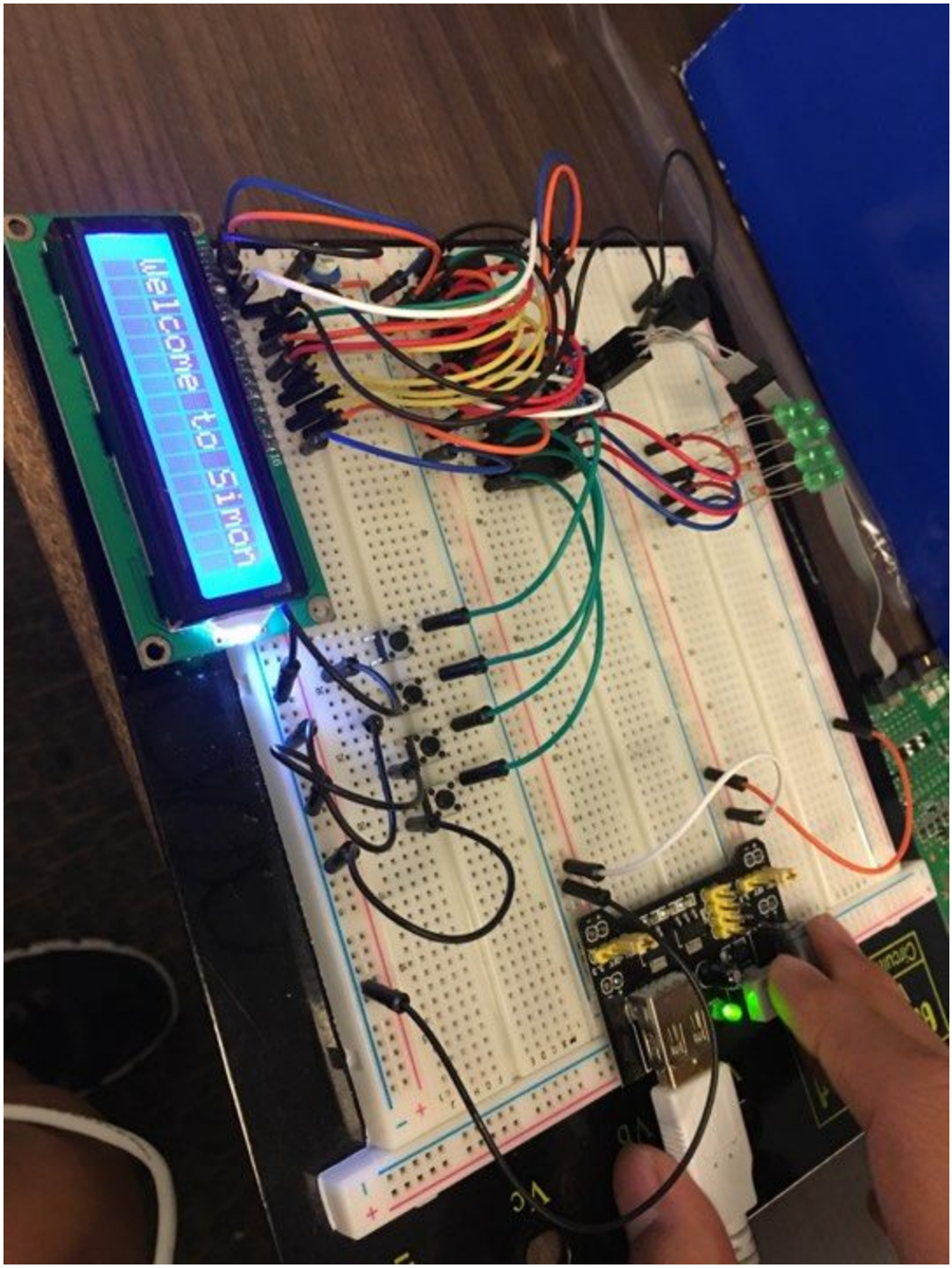
I used AVR Studio 6, an ATmega1284 microcontroller with 4 330 ohm resistors and a potentiometer that helps display what I want on the LCD. I use the ISP adapter to implement the program onto the ATmega1284 microcontroller and from there, the program is being read outputting the values I needed and waited when an input from the player was required. I set the speaker on PORTB6 on the ATmega1284 microcontroller and used the function so that only specifically PORTB6 can be used to output a sound. My inputs are set for PORTA where the buttons are and all of PORTB and PORTC are used for my outputs which are the LEDs, speaker, and LCDs. PORTD is set for the data control lines that are being used by the LCD screen.

For the bonus part of the lab, I set an array of a certain amount of frequencies so that each button can output a different frequency. Whenever a button is pressed, each button has its own output frequency so that they are distinguishable among each other. When the player loses the game or wins the game, sounds are outputted and then reset after the game. The program has it so that I only output the sound and LED for two seconds and wait for the next input of the button once the button is released.

From this project, I gained a better understanding of button debouncing because this was the biggest obstacle I faced when creating this program. This caused me a lot of trouble as my button inputs were overlapping and causing an automatic loss in the game because it keeps reading the same inputs multiple times. I was missing a state where I had to release the button and wait for the next input and read that. Another obstacle I faced was the compare array and outputting the input values. Compared to the beginning where our random outputs could be outputted in a loop, the inputs had to be constantly overwritten and then outputted out again. I was having an issue where the inputs would sometimes never even output and this made me release that my compare arrays were having an issue as well since the values from the previous inputs were not overwritten so I just got an automatic loss from this as well. I also implemented a reset to the game so that no matter if the player wins or loses, they can keep playing until they turn the power off.

http://www.cplusplus.com/reference/cstdlib/ This library was to access the rand() function http://www.atmel.com/webdoc/avrlibcreferencemanual/group__util__delay_1gad22e7a36b80e2f917324dc43a425e9d3.html This library was to access the _delay_ms() function instead of using a timer function

https://youtu.be/xfWVa5ADUe8 This is the video for the demo of the Simon game discussing the instructions and guidelines to how to win or lose.


Welcome to Simon