Alec Pflaumer
CS 120B - B21
Professor Kelly Downey
31 August 2017

Final Project - Simon Game

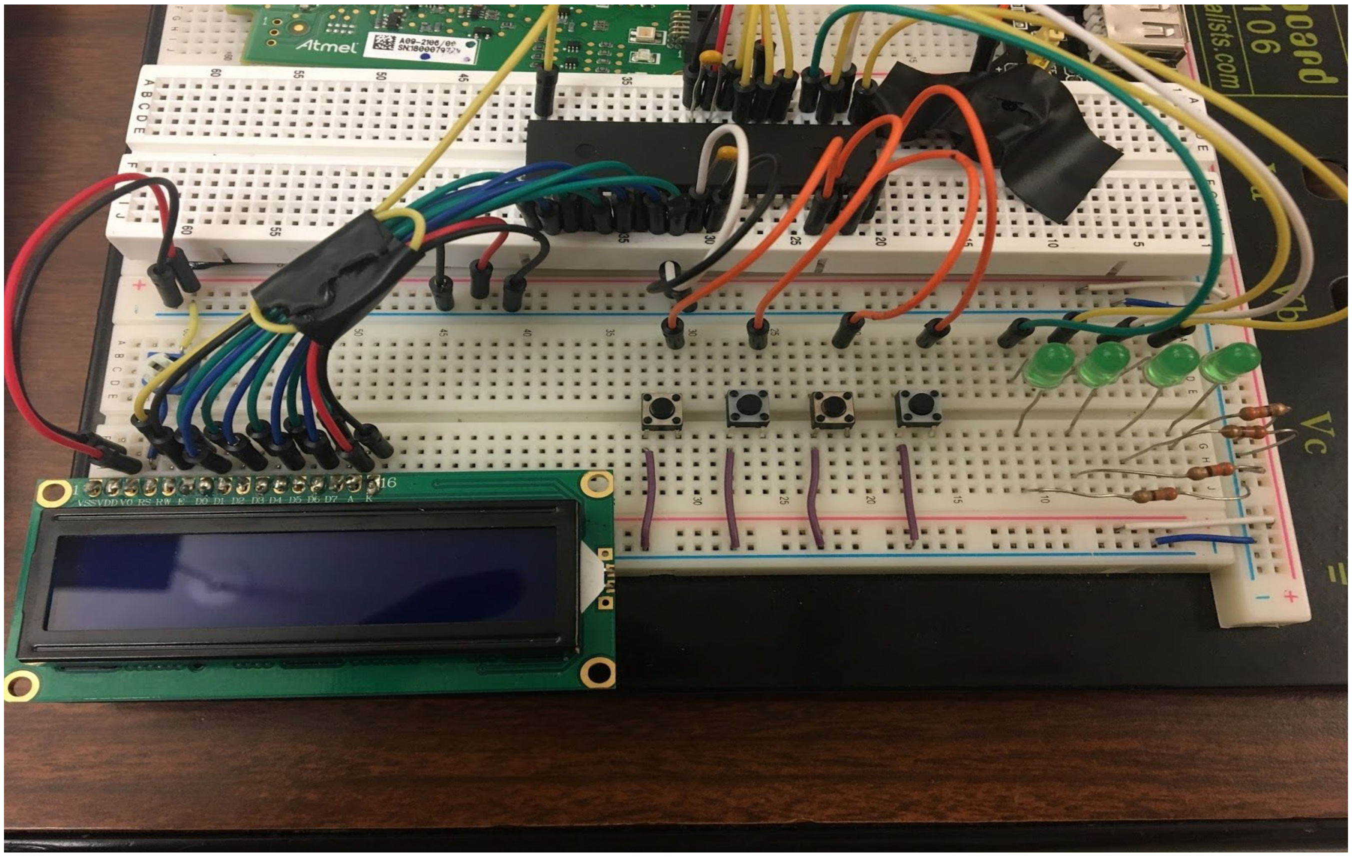
Github and Video Demonstration Links

GitHub: https://github.com/AlecPflaumer/CS_120B_Final_Project

YouTube: https://youtu.be/oDh5_MYNC7A

Project Description and User Interface

The ultimate goal of this project was to replicate the electronic game Simon with an ATmega1284 programmed in Atmel Studio. As you can see below, there are 4 LEDs lined up on the right side of the breadboard that act like the colored lights encircling the Simon interface and four buttons to the left of those LEDs to mirror the setup of the four LEDs. These are the buttons the user would use to attempt to replicate the pattern played on the LEDs. The LCD screen on the left side is there to greet the user, display the wins and losses of the session, and tell user when they have won or lost and then prompt them to start a new game. There is a small buzzer covered in electrical tape in the top right to sound in accordance with the LEDs.



Methods

High-level model with several concurrent state machines:

- (Relevant) Global Variables: **pattern[9]**, **pIndex**, **numWins**, and **numLosses** (all unsigned chars)
- Synchronous State Machines:
    1. <u>NewGame</u>
        a. Performs its duties at the start of every new game, i.e. upon powering the system on and directly after winning or losing a round and restarting.
        b. Sets up **pattern** to hold 9 random numbers from 1-4 corresponding to the four different LEDs that can light up.
        c. Sets **pIndex**, which keeps track of how far into the pattern the user has guessed correctly so far, to 0.
        d. If it's the first game (upon powering on), then the system will prompt the user to press a button to start. If it's any subsequent game, the user has already been prompted to press a button to start a new game from the <u>EndGame</u> SM.
        e. Once the user presses and releases a button, the system displays the number of wins and losses on the LCD screen and goes back to its Idle state and tells <u>OutputSeq</u> to perform its duties.
    2. <u>OutputSeq</u>
        a. Lights up the LEDs and sounds buzzer according to **pattern** and up until **pIndex**, ignoring input along the way.
        b. Once it's done it returns to its Idle state and tells <u>InputSeq</u> to begin.
    3. <u>InputSeq</u>
        a. Waits for the user to press a button.
        b. While the user is pressing a button it lights up the corresponding LED and sounds buzzer.
        c. Steps a-c repeat until the user has correctly imitated **pattern** up until **pIndex** or the user has entered the wrong button and lost.
            i. If the user has won (which increments **numWins**) or lost (which increments **numLosses**) the game the SM returns to its Idle state and tells <u>EndGame</u> to start its processes.
            ii. If the user has correctly entered up until **pIndex** the SM returns to Idle state, increments **pIndex**, and tells <u>OutputSeq</u> to start again.
    4. <u>EndGame</u>
        a. If the user has won the game then the LEDs light up in a celebratory pattern and tell the user they have won and to press a button to play again.
        b. If the user has lost the game then LEDs flash and tell the user they have lost and to press a button to play again.
        c. Once the user has a pressed a button <u>EndGame</u> returns to its Idle state and tells <u>NewGame</u> to begin.

As you can see, the game uses a "relay race/passing-of-the-baton" system to operate; that is, only one of the above synchronous SMs will be running at a time, and once each is done it puts itself into Idle mode and tells the next SM to begin. Further commentary on this subject is in my conclusion.

Problems Encountered and Solutions

One problem I encountered early on was that, after writing all my code, I realized I wasn't sure how to put my ATmega in sleep mode while while it was in the "while(1)" loop. After doing some research I found out that I simply needed to `#include <avr/sleep.h>` and use its functions to use sleep mode. Finding out how to use the library's functions led me to find out in the ATmega1284 datasheet that there are actually several different kinds of sleep modes to choose from that all had varying degrees of power conservation! I used `SLEEP_MODE_IDLE` out of fear of somehow rendering my microcontroller unusable.

Another major issue I had was that at one point when the user would win or lose the system would skip straight to the win or lose scenarios of EndGame without displaying what the user pressed and then skipping through the end-of-game LED sequence and LCD screen message SUPER quickly since the user would still have the button pressed from when they were imitating the last part of the LED sequence. This required me to significantly alter a few of my states in InputSeq and is a problem I wish I caught before writing out all the code, where it was more difficult to see where the issue was coming from and and how to fix it.

Conclusions and Improvement Ideas

Upon finishing this project I can say that there are several things I would do to improve the system and a few lessons I have learned.

If I were to keep developing this project in the future the most significant change I would make would be using some sort of alternate method to the "passing-the-baton" system I mentioned in my methods sections. It ended up working fine for this project but I can see how easy it would be to make a mistake and have more than one SM running at the same time when they shouldn't and yielding unexpected and unwanted results. A better system could use more master-servant relationships and handshake functionality. Perhaps this would best be done if I had a master-SM that controlled which SMs were to do what at what time.

In a similar vein, I believe I could clean up my code a bit by making the LCD screen and the buzzer have their own SMs since they aren't necessary to have the rest of the system working and detract from the reader's attention. It might actually eliminate excess code since the buzzer always only plays when an LED is lit up and determines which sound to play based off of what LED is being lit up.

A major improvement would be figuring out a good way to seed `srand()` at the start of the program since right now it just uses `srand(0)`, which yields the same patterns every time you power on the system instead of having truly random ones. Unfortunately, the ATMega1284 has no way to keep time, which is the easiest and most common way of seeding random functions.

The only major features I believe would be easy to add to improve the user's experience would be letting the user choose the difficulty level, i.e., how big the pattern to imitate is.

Overall the project was a success, but I certainly learned a thing or two about rushing into coding, why we should use incremental design, and how important it is to avoid accumulating technical debt.