Building the Game *Simon* with Arduino

Sam Frank
(Dated: March 15, 2023)

I. INTRODUCTION

The game *Simon* is an electronic hand-held toy that tests the player's memory and reflexes. It features an ever-growing sequence of flashing lights that the player must correctly repeat by pressing the appropriate colored buttons (Figure 1). Each button is associated with a unique tone that plays when it lights up or is pressed by the player. As the player correctly remembers the sequence, more lights are added to the sequence, making it increasingly difficult to recall. The game continues until the player fails, either by taking too long or by pressing the wrong button in the sequence, or wins by correctly inputting the entire sequence (some versions of the game may continue indefinitely, with the difficulty increase coming from the speed of the sequence increasing rather than adding more lights to the sequence).

Figure 1: A commercial version of the game *Simon*. Players attempt to remember the sequence of lights and tones by pressing buttons in the correct order.

As an example for how a typical *Simon* game plays out, suppose that one version of the game features four colors: green (G), red (R), yellow (Y), and blue (B) each with four distinct tones (T1, T2, T3, and T4, respectively). Then one possibility of the game could play out as follows:

1. Simon: R(T2)
2. Player: Presses R(T2)
3. Simon: R(T2), G(T1)
4. Player: Presses R(T2), then G(T1)
5. Simon: R(T2), G(T1), G(T1)
6. Player: Presses R(T2), then G(T1), then again G(T1)
7. Simon: R(T2), G(T1), G(T1), B(T4)
8. Player: Presses R(T2), then Y(T3)
9. **Game over** (player pressed the wrong button)

As *Simon* involves multiple components working together, such as flashing LEDs, different tones, and internal rules and logic, it's an ideal choice for a final electronics project. Here we use an Arduino and 8 Ω speaker in order to construct a demo version of the game. We only use three buttons here instead of four (the added complexity from adding a fourth button is minimal), and we limit the sequence length to 10 buttons instead of allowing it to continue indefinitely. We also include various sound effects to enhance the player's experience. For instance, a buzzer sound plays when the player presses the wrong button or takes too long, and the coin sound from *Super Mario* sounds when the player presses the correct button. Additionally, if the player successfully inputs the entire sequence, the vintage level complete song from *Super Mario* plays (which is similar to what is heard when Mario touches the end level flagpole).

II. BUILDING *SIMON*

The construction of the game involved several stages. First, we had to build the circuit that connected the LEDs to the Arduino, which illuminated the lights as the sequence played for the player.

Second, we had to create the drive circuit to power the breadboard's speaker from the Arduino. We also had to program the speaker to play the correct tones when the corresponding lights flashed.

Third, we constructed and programmed the buttons that the player presses. This involved ensuring that the buttons lit up when pressed, played the correct tone when pressed correctly, and played the buzzer sound when pressed incorrectly.

Finally, we programmed the end-of-game protocol that determines when and how to restart the game after the player either wins or loses.

By completing these four stages, we successfully constructed a functional version of the Simon game using an Arduino and a breadboard.

Figure 2: Schematic of the LED circuit. Here "5 V" stands for the Arduino.

A. Building the LED Circuit

The LED circuit was built by connecting a 220 Ω resistor and LED in series into the digital pins (in this case, we used 2, 3, and 10 for each of the three LEDs). We grounded the circuit with respect to the Arduino's ground.

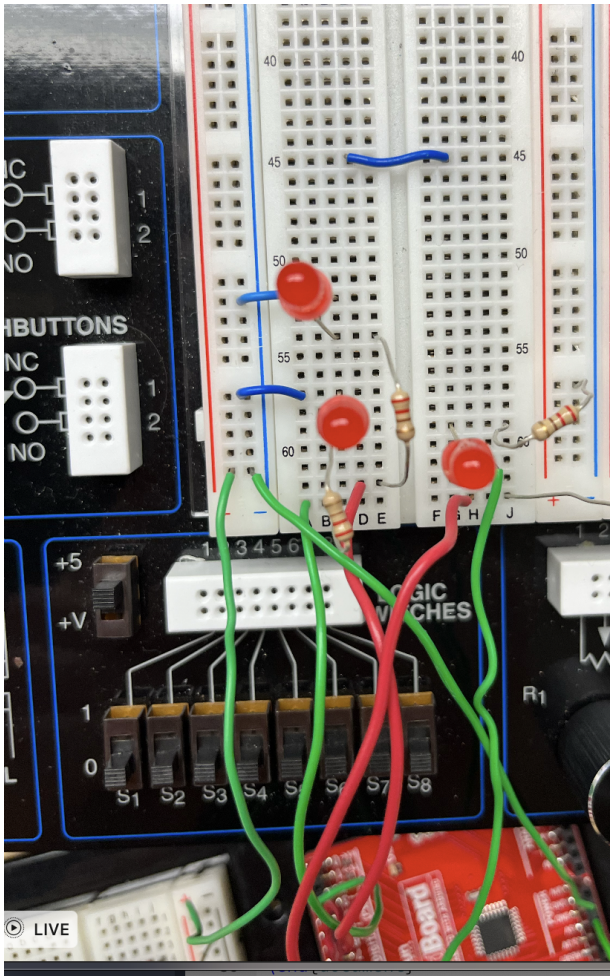

Figure 3: The LED circuit component of *Simon*.

Programming the LEDs to light up was a simple application of elementary commands in Arduino (e.g., `digitalWrite`). To generate the random sequence, an array `rand_sequence` is used to store a random sequence of 0's, 1's, and 2's. The entire array (of length `len_sequence`) is stored, and the elements of the array are sequentially outputted using a variable `round_num` (see code linked in the Appendix).

B. Building the Drive Circuit to Power the Speaker

Because the Arduino cannot power the speaker itself, a drive circuit is used to amplify the signal so that it can be heard over the speaker.

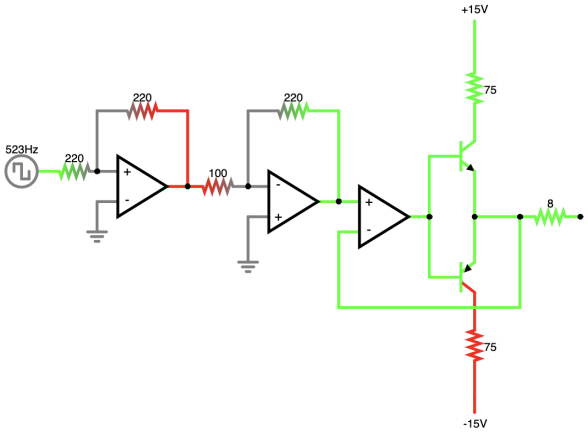

Figure 4: Schematic of the drive amplifier circuit used to power the speaker. Here, the "523 Hz" square wave signal represents the Arduino, and the "8 Ω" represents the speaker.

The subcircuit defined from the second op-amp to the speaker (represented by the 8 Ω element) is referred to as the control drive. It consists of a summing amp followed by an op-amp with push-pull output stage. The collector resistors were chosen such that the maximum current (200 mA) flows through the transistors (15 V / 0.2 A = 75 Ω). Note that the summing amp has gain greater than unity (the value of the 100 Ω resistor was chosen via trial and error), and the push-pull output works as a current booster. The other subcircuit (the first op-amp) is a proportional op-amp that ensures 0 phase shift between the input and output. Without the proportional op-amp, the output has a phase shift of π relative to the input.

To program the speaker, we can simply use the built-in function `tone` that outputs a 5 V square wave at a specified frequency. In this case, we chose frequencies 440 Hz (A4), 523 Hz (C5), and 698 Hz (F5).

C. Integrating the Buttons

To integrate the buttons into the physical circuit, we simply connected one end to a digital pin (in this case 7, 8, and 5), and the other end to ground (with a 10 kΩ resistor that is used in order to more quickly and efficiently

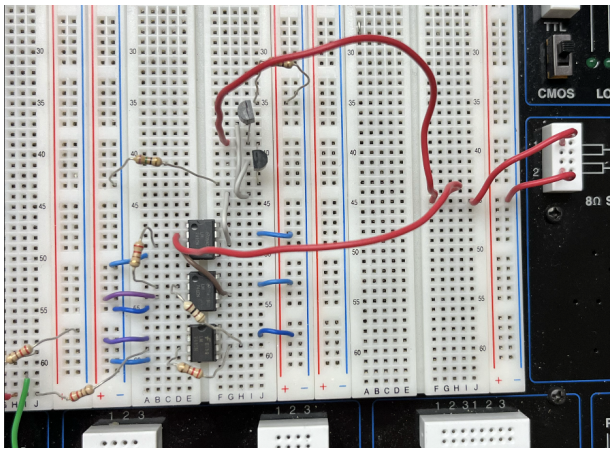

Figure 5: The speaker drive circuit component of *Simon*.

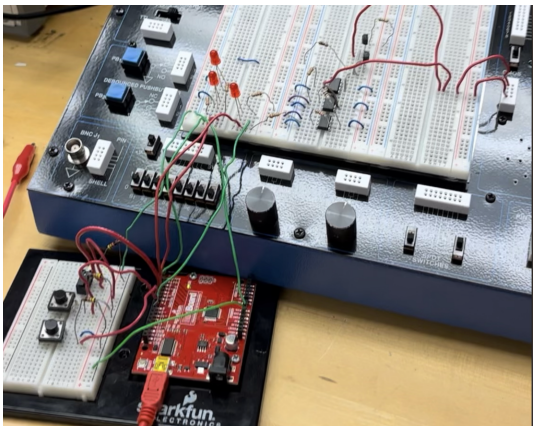

Figure 6: Finished *Simon* demo product. We see all three main components of the circuit: the buttons in the bottom left, the LED's in the top left, and the drive speaker circuit in the top right.

ground the button response). Causing the button to light up the correct LED was simply a matter of writing the correct conditional statements. We used the Booleans `button_state0`, `button_state1`, and `button_state2` to represent the state of each button pressed. Through trial and error to determine the optimal behavior of the buttons, we found that allowing each button to light up the LED for 0.25 s and having 0 s of delay in between each button press resulted in fewer erroneous game overs.

D. Designing the End-of-Game Protocol

At this stage, we have a functional *Simon* game, but it doesn't yet know how to handle incorrect button presses or the correct repeating of the entire sequence. We used a Boolean `player_lost` and put the LED/speaker and button code segments within a `while (player_lost == LOW)` loop. If a button is pressed incorrectly, we set `player_lost` to `HIGH` and `round_num` to 1. Switching `player_lost` to `LOW` resets the main `while` loop, and putting a conditional `if (round_num == 1)` statement at the start of the loop enables us to reset the variable `rand_sequence` into a new random sequence. See attached code for details. Note that an incorrect press is treated the same as a timeout (set arbitrarily to 4 s without any button input from the player). If the player makes it through the entire sequence and inputs it correctly, then the game starts over with a new random sequence.

To ensure the player knows when a button has been pressed correctly or incorrectly, we used unique sounds for each button press. For a correct button press, we play the coin pick-up sound from *Super Mario*; for an incorrect button press, we play a loud buzzer sound; for correctly inputting the entire sequence at the end of the game, we play the vintage end-of-level song from *Super Mario* (the song that plays when Mario touches the flagpole).

Figure 7: Commercial version of the game *Bop It*. Implementing different buttons or player responses similar to *Bop It* would be an interesting way to improve upon this project.

III. FINISHED PRODUCT AND CONCLUSIONS

After implementing each stage of the project as explained above, we have a working *Simon* demo game.

Moving forward, it would be interesting to see if we could implement different kinds of buttons or player responses, similar to the game *Bop It*.

Visit this public GitHub repo for the code and demo video. To view the video, click "View raw" and download the file. The video first demonstrates successful completion of a length-five sequence. Then, a new sequence starts over, and we show three possible ways for the player to fail: by pressing the wrong button, by waiting longer than 4 seconds in between two button presses, and by waiting longer than 4 seconds after the initial playback of the sequence (before any button is pressed).