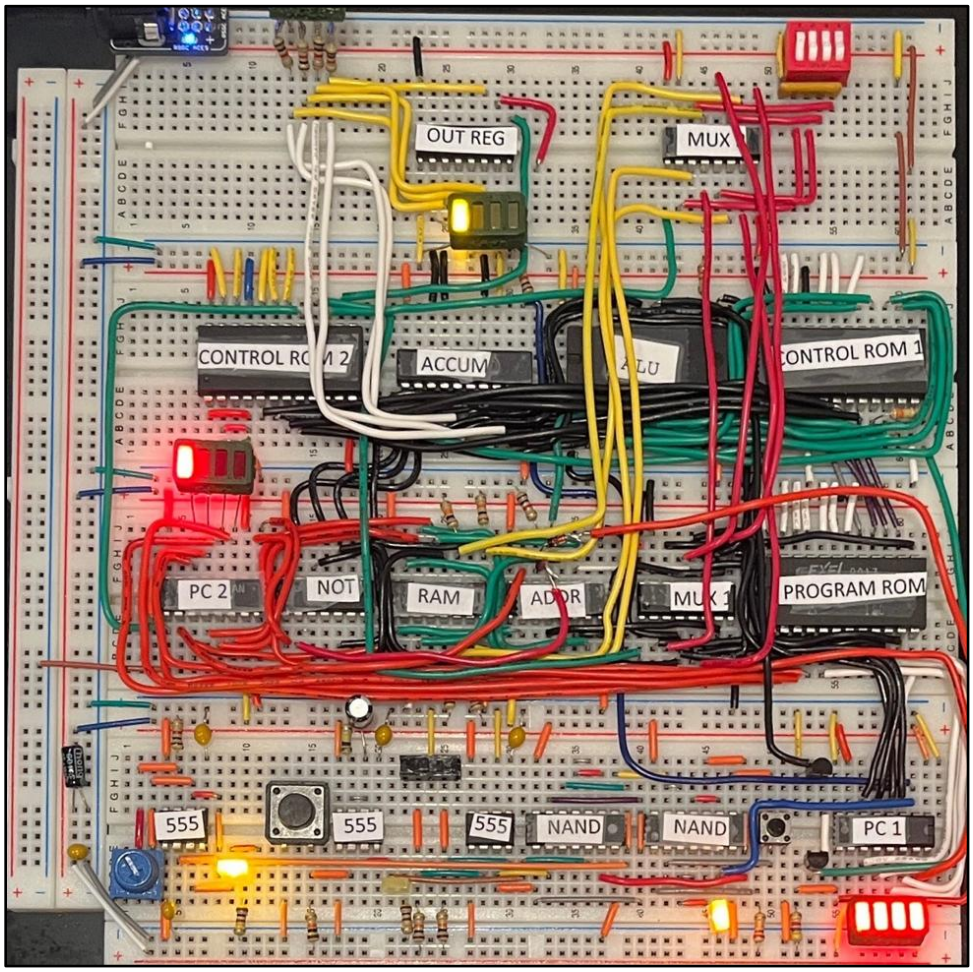
# Design Engineering Report




Course: Introduction to Computer Science
Code: ICS2O-E, ICS3U-E, ICS4U-E
Author: Rohan Jamal
Date: Wednesday, November 5, 2025

# Table of Contents

# ICS2O-E

# Project 1.1 Voltage H-Bridge

## Purpose

In addition to exhibiting the voltage-dividing characteristics of a potentiometer, the purpose of the Voltage H-Bridge is to demonstrate the way that current flows between potential differences.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#hBridge

Project schematic: https://crcit.net/c/81605d037990431b9416081eb7c51ff4

## Procedure

The Voltage H-Bridge circuit employs the *voltage-dividing* properties of a potentiometer. A voltage divider consists of multiple resistors in series with an intermediary point that can be accessed to produce a particular fraction of the supply voltage. Kirchhoff's voltage law states that the sum of the voltages in a circuit is zero; therefore, when connecting two resistors in series without additional load, they consume all the supply voltage. When connecting two resistors in series with an output (see Figure 1) the remaining usable voltage can be described as $V_{out} = \frac{V_{source} \times R2}{(R1 + R2)}$. Both fixed and variable voltage dividers exist; a fixed voltage divider consists of two fixed resistors in series while a potentiometer can act as a variable voltage divider. In the latter configuration, the potentiometer acts as two variable resistors in series, with the third lead acting as a tappable point between the two resistors (see Figure 2). As the potentiometer is adjusted, the resistance before the third lead changes, and the resistance after the third lead changes inversely, adjusting the output voltage.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 4 | 470 Ω Fixed Resistor |
| 2 | 10 KΩ Potentiometer |
| 2 | 5mm Bicolor LED |
| 1 | 9V Power Source (Battery) |
| 1 | Breadboard |
| 1 | Stripboard |
| 1 | Terminal block |
| ~ | Wires |


Figure 1. Fixed voltage divider


Figure 2. Variable voltage divider

The Voltage H-Bridge circuit consists of a *bicolor light-emitting diode* (bicolor LED) spanning one fixed and one variable voltage divider (see Figure 3). A bicolor LED contains two LEDs in inverse parallel. Which LED is active depends on the direction of current because diodes are *polarized components*. Polarized components are parts that can only be connected in one direction. In the Voltage H-Bridge circuit, two voltage dividers are used to change the direction of current and color of the bicolor LED. Current always flows from the higher potential to the lower potential.

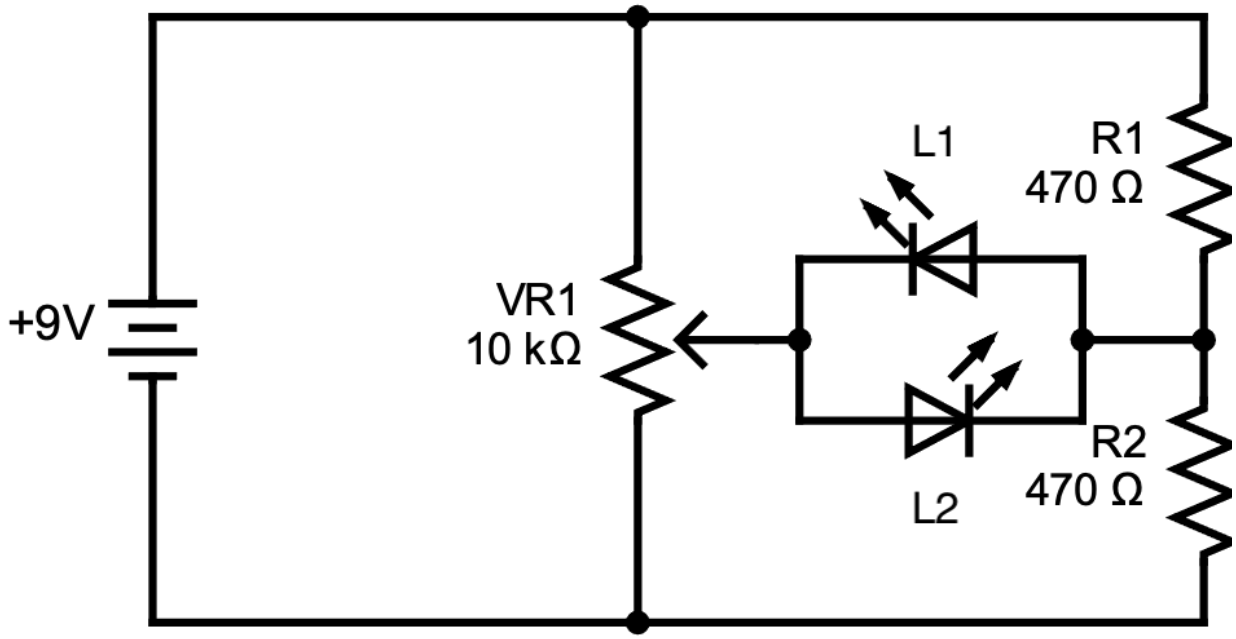
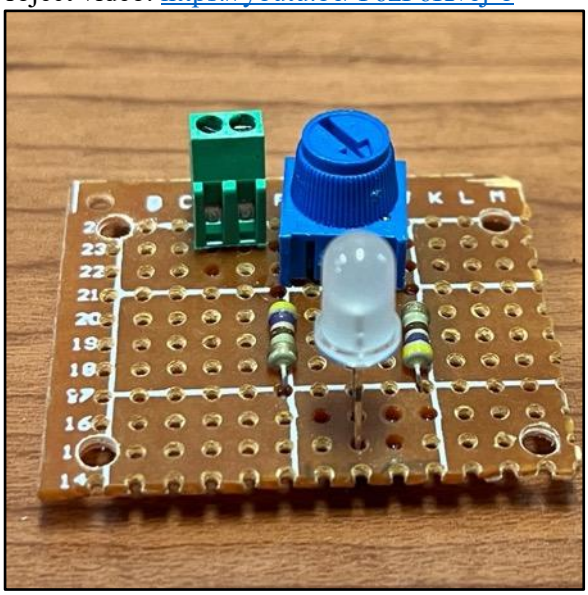



Figure 3. Voltage H-Bridge

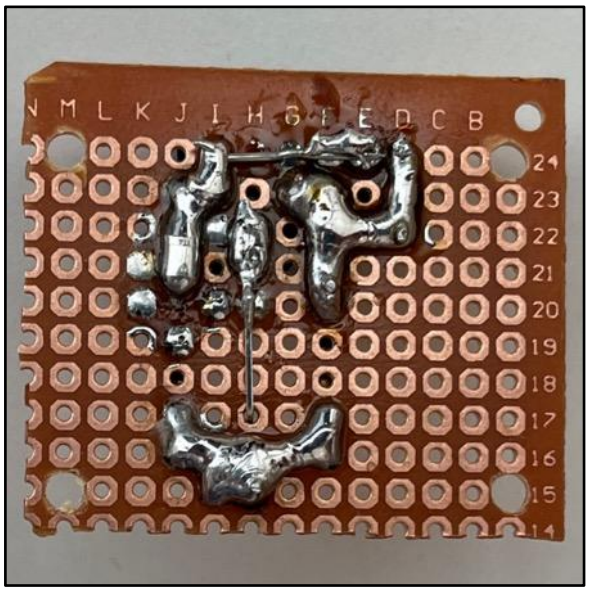
The voltage between R1 and R2 can be expressed using *Ohm's Law* ($V = IR$) as $V_{out} = \frac{9V \times 470\,\Omega}{940\Omega}$ which is equal to 4.5 V; therefore, while the voltage coming out of the potentiometer is greater than 4.5 V, current flows from the variable voltage divider to the fixed voltage divider, illuminating L2; while the voltage is less than 4.5 V, however, current will flow in the opposite direction, illuminating L1. The voltage coming out of the potentiometer changes from 0 V to 9 V as the potentiometer is adjusted.

## Media

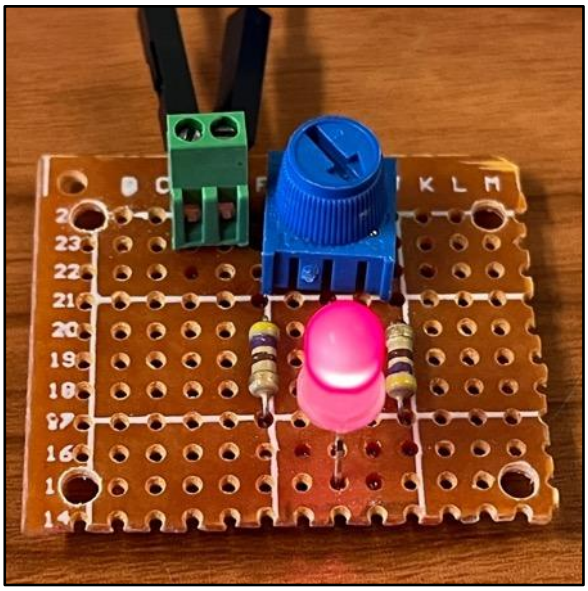
Project video: https://youtu.be/Y6zF6Kvej-o



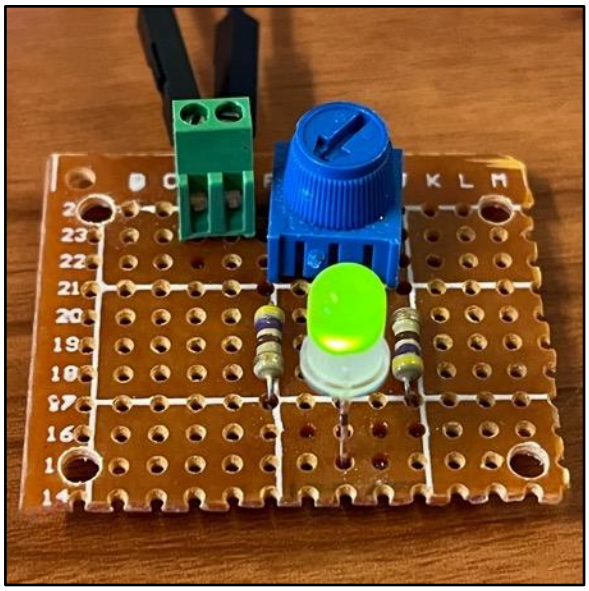
Top of circuit (Stripboard version)



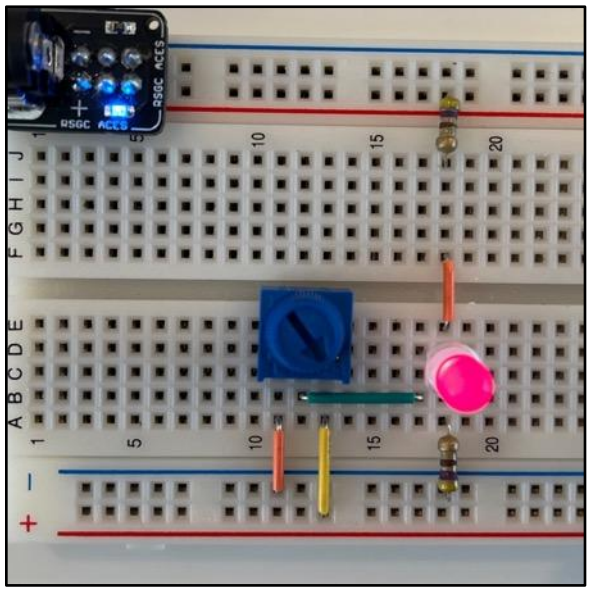
Bottom of circuit (Stripboard version)



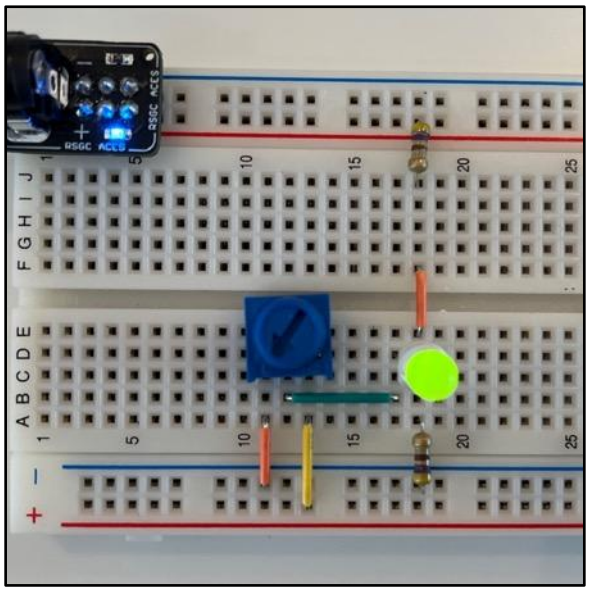
Potentiometer: C → A (Stripboard version)



Potentiometer: C → B (Stripboard version)



Potentiometer: C → A (Breadboard version)



Potentiometer: C → B (Breadboard version)

## Knight Light

### Purpose

The purpose of the Knight Light is to demonstrate how some types of voltage dividers and resistors can be used in conjunction to change the brightness of an LED based on ambient lighting conditions.

### Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#hBridge

Project schematic: https://crcit.net/c/685b7bc5725c4a7993dd24905605d33a

## Procedure

Much like the Voltage H-Bridge circuit, the Knight Light makes use of an LED spanning one fixed and one variable voltage divider (see Figure 1). The fixed voltage divider has a potential of $\frac{9V \times 470\,\Omega}{940\Omega} = 4.5\,V$ and consists of two fixed resistors in series. The variable voltage divider consists of a potentiometer and a *light-dependant resistor* (LDR). An LDR is a component where light and resistance have an inverse relationship. As ambient light increases, the resistance of the component decreases. While light is hitting the LDR, it has a low resistance, causing the variable voltage divider to have an output voltage less than 4.5 V. The LED blocks the current, exhibiting its *diodic properties*. A diodic property is the characteristic of a component that only allows current to flow in one direction by blocking current flowing in the opposite direction. The current must flow from the variable voltage divider to the fixed voltage divider in order to match the polarization of the LED. When covered, the LDR has a high resistance, driving the output voltage of the variable voltage divider up, above 4.5 V, allowing current to flow through and illuminate the LED.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 2 | 470 Ω Fixed Resistor |
| 2 | 10 KΩ Potentiometer |
| 1 | Light-dependant resistor |
| 1 | 5mm LED |
| 1 | 9V Power Source (Battery) |
| 1 | Breadboard |
| ~ | Wires |

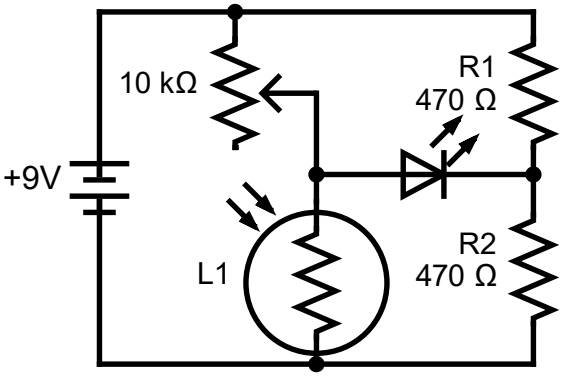
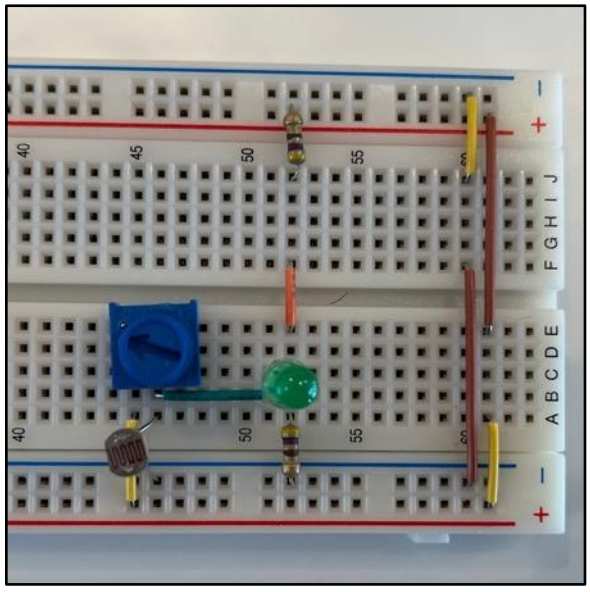
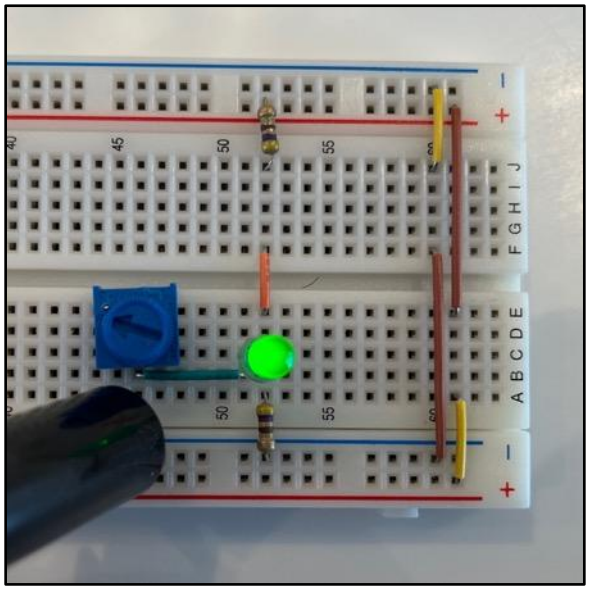



Figure 1. Knight light

## Media



LDR low resistance



LDR high resistance

## Reflection

This project has been very interesting for me. I remember Mr. D'Arcy telling the class, "*It's only after 10-15 hours of working on your DER that you know the true purpose of the assignment,"* and I thought, "*There's no way that anyone is spending that long on this project!"* I have never been so wrong. Throughout the past few days, I have put in the 10-15 hours, and when looking over my DER, I have realized that it is well worth the commitment. This project has put me in the driver seat and taught me about voltage dividers in much more depth than would have been possible from the passenger seat. The build process was mostly smooth for me, aside from some minor hiccups, such as having to resolder a blown LED. Time management has not been a major problem during this project, which is something I am happy about. It is not going to be a late night working on my DER, submitting it just minutes before midnight. At least that is what I thought when originally writing the reflection; eventually, the video proved to take longer than expected. In hindsight, however, it could have been worse; I could have really been scrambling until minutes before, or even submitted it late. Overall, I am quite happy with my report and I am looking forward to writing the next one.

# Project 1.2 Joule Thief

## Purpose

In addition to exhibiting the fast-switching capability of the transistor, the purpose of the Joule Thief is to demonstrate how inductors can serve as voltage boosters.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#joule

Project schematic: https://crcit.net/c/c59f3811ddcd4e31b1fb176c15f78af3

## Procedure

The Joule Thief circuit makes use of the voltage-boosting properties of inductors. Two or more inductor coils can be used to increase or decrease the output voltage of a power supply. An inductor is a *passive component* that resists changes in current passing through it. A passive component is a component that does not consume supply voltage, but alters it.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | 1 KΩ Fixed Resistor |
| 1 | 0.5-1.5 V AA Battery |
| 1 | 2.8 V White LED |
| 1 | Toroid Ring |
| 1 | 2N3904 NPN Transistor |
| 1 | SPDT Slide Switch |
| 1 | AA Battery Holder |
| 1 | Breadboard |
| ~ | Wires |

Inductors store energy in magnetic fields. Faraday's law of induction describes how any current running through a wire produces a magnetic field. An inductor takes advantage of this property. It consists of a conductor wound in a loop, compressing the generated magnetic field into a smaller area (see Figure 1). This creates a stronger magnetic field. Faraday's law of induction also describes how a changing magnetic field can induce a voltage in a wire.

When a current is passed through an inductor, it, at first, offers high resistance, converting the electrical energy to a magnetic field. The resistance of the inductor slowly decreases, until it reaches its maximum. Once its maximum is reached, the inductor no longer offers significant resistance; it behaves as a normal conductor would. When the power source is removed, the inductor continues to resist change in electrical current; its magnetic field begins to collapse, releasing the stored energy as electricity.

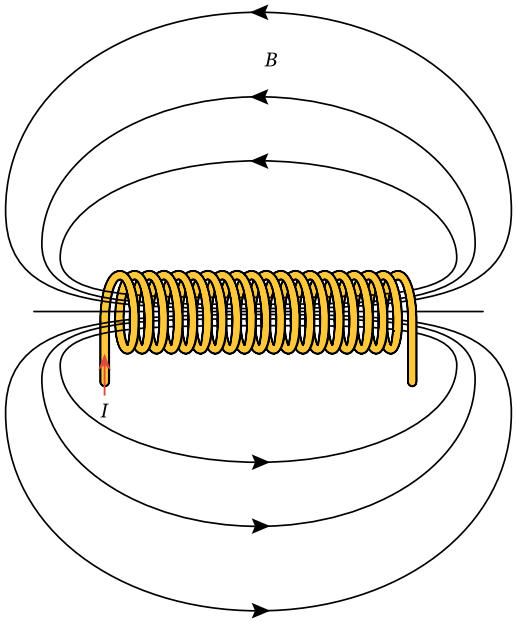



Figure 1. Inductor and magnetic field

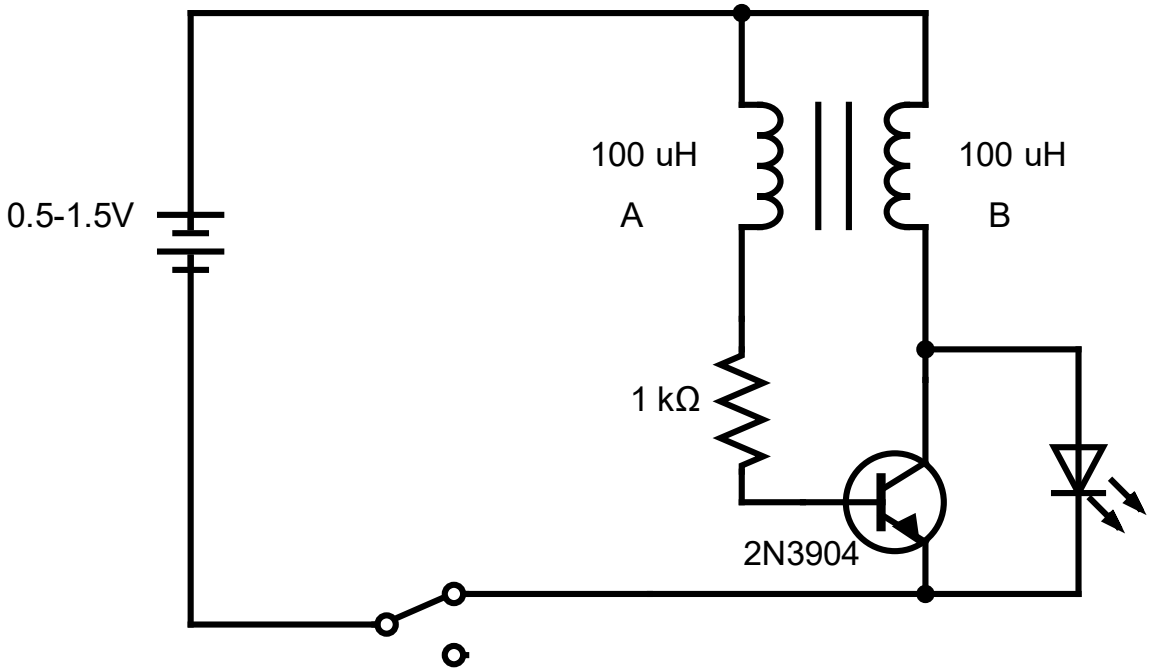



Figure 2. Joule Thief

The Joule Thief circuit (see Figure 2) consists of two inductors, a transistor, and a *light-emitting diode* (LED). When switched on, a small amount of current flows through inductor A and the resistor to the base, partially activating the transistor. With the collector and emitter partially connected, current begins to flow through inductor B, increasing the strength of its magnetic field. As the strength of the magnetic field around inductor B increases, a magnetic field is induced on the other inductor, causing the base of the transistor to open more. This causes an even stronger magnetic field to form around inductor B, leading to a larger magnetic field induced on inductor A. This process continues until the transistor is *saturated*. A saturated transistor is fully conductive, meaning that maximum current can flow through it. Faraday's law of induction states that only a changing magnetic field can induce a voltage in a conductor, meaning that once the magnetic field of inductor B becomes *static*, it stops inducing electricity in inductor A. A static magnetic field refers to a magnetic field that remains the same. Without electricity being induced in inductor A, the transistor starts to close, decreasing the electricity available to inductor B. This continues until the

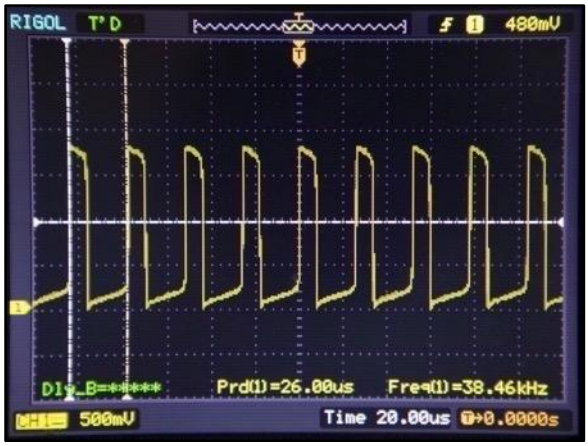



Figure 3. Joule Thief Oscilloscope Readings

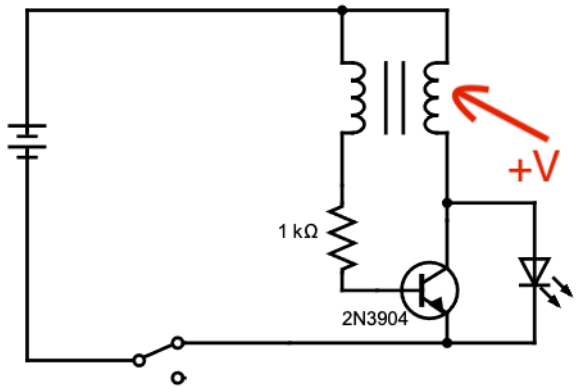



Figure 4. Source of Voltage Spike

transistor is nearly completely closed. Once the transistor base closes, the magnetic field storing energy in inductor B collapses, leading to a voltage spike across the LED, resetting the whole circuit (see Figure 4). This causes the process to start again from the beginning.

A Joule Thief circuit repeats this process at a rate upwards of 30,000 Hz, causing the LED to flash at this rate (see Figure 3). *Persistence of vision* is an optical illusion that occurs when the human eye retains an image for a short period of time after the source of that image is removed. This effect makes the LED appear to stay on constantly, despite the fact that it is flickering.

## Stripboard Version

Once the Joule Thief prototype has been built on breadboard, the next step is to build it on a stripboard. A stripboard consists of holes with copper pads, spaced 0.1 inches apart. Components can be soldered anywhere on the board, and connections are made with either wires on the top, or solder joints on the bottom. Since no connections exist prior to soldering, a stripboard circuit can be much more compact. The stripboard Joule Thief functions the same as the breadboard version does, however, unlike the breadboard, it can be carried around and is in a permanent configuration; everything is soldered down, meaning that it will not fall off.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | 1 KΩ Fixed Resistor |
| 1 | 0.5-1.5 V AA Battery |
| 2 | 100 µH Fixed Inductor |
| 1 | 2.8 V White LED |
| 1 | 2N3904 NPN Transistor |
| 1 | SPDT Slide Switch |
| 1 | AA Battery Holder |
| 1 | Terminal Block |
| ~ | Solder |
| ~ | Wires |

## Finished PCB

The final step in the Joule Thief project is soldering the finished *Printed Circuit Board* (PCB). This has a much more finished look than a stripboard circuit does, as it is custom made for the application. It is also much simpler to setup. While the PCB Joule Thief functions identically to the stripboard Joule Thief, the PCB comes with all connections already made. This means that once each component has been soldered onto the PCB, no additional steps are needed to make the appropriate connections.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | 1 KΩ Fixed Resistor |
| 1 | 0.5-1.5 V AA Battery |
| 2 | 100 µH Fixed Inductor |
| 1 | 2.8 V White LED |
| 1 | 2N3904 NPN Transistor |
| 1 | SPDT Slide Switch |
| 1 | AA Battery Holder |
| 1 | RSGC ACES Joule Thief PCB |
| ~ | Solder |

## Media

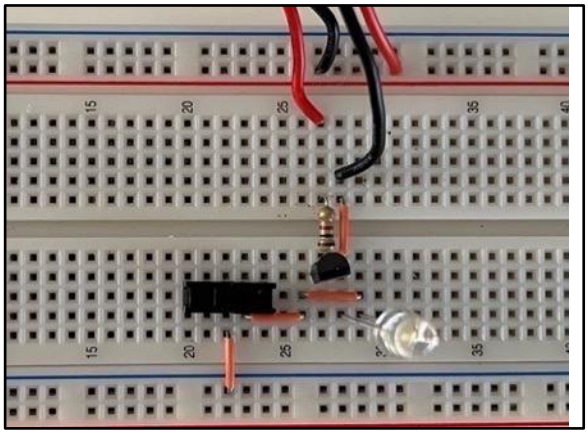
Project video: https://youtu.be/3ehgFpXpgIw


Bottom of circuit with battery (PCB)

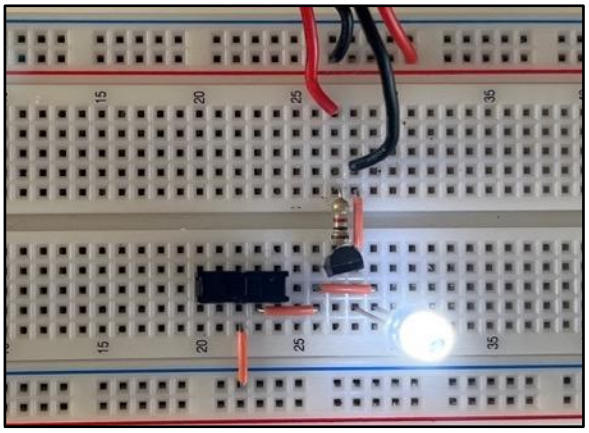
Bottom of circuit without battery (PCB)


Switch: GND → Floating (PCB)

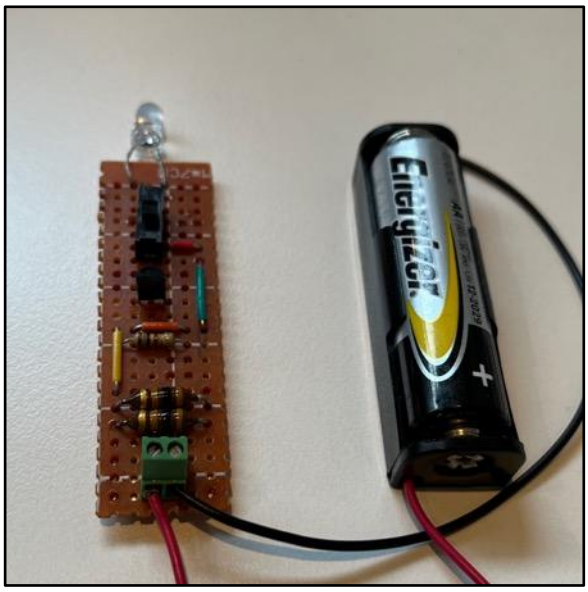

Switch: GND → Emitter and LED (PCB)



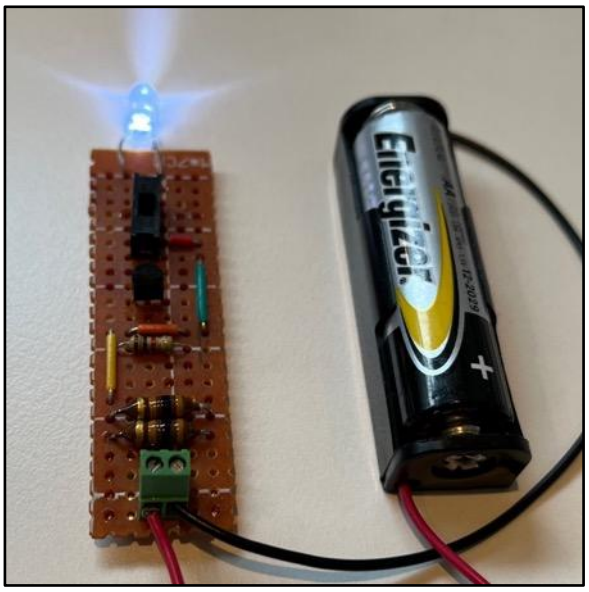
Switch: GND → Floating (Breadboard)



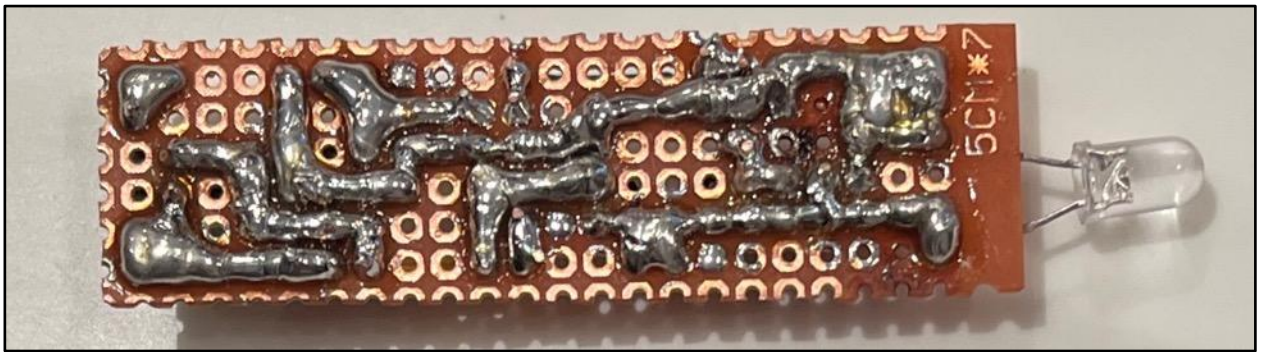
Switch: GND → Emitter and LED (Breadboard)




Switch: GND → Floating (Stripboard)




Switch: GND → Emitter and LED (Stripboard)



Bottom of circuit (Stripboard)

## Reflection

I have found this project to be extremely hands-on, which is something I like about it. Everything from winding my own inductor, to soldering a stripboard version of the circuit had its hiccups, like when I could not figure out why my toroid was not working, or when I could not get my stripboard circuit to light up. Eventually, I did overcome these obstacles. I guess that is what engineering is truly about: finding the problem, brainstorming potential solutions, eliminating the problem, then moving on to the next one. This is something that I enjoy and look forward to doing in the future. This time around, I have been pretty good about time management. Everything is done Tuesday, which means it's not going to be a late Wednesday night, with me scrambling to hit "send" at 11:59. Overall, I am very pleased with this project, and I am looking forward to the next one.

# Project 1.3 1-Bit Magnitude Comparator

## Purpose

In addition to demonstrating the behaviour of different logic gates with varying inputs, the purpose of the 1-Bit Magnitude Comparator is to exhibit how digital logic circuitry can be used to put different magnitudes in order.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#magnitude

Project schematic: https://crcit.net/c/58e81855290e4ca390e0bf525883dfc0

https://byjus.com/gate/difference-between-combinational-and-sequential-circuit/

https://www.javatpoint.com/sequential-circuits-in-digital-electronics

## Theory

Nearly all machines have 3 basic sections: input, processing, and output. The 1-Bit Magnitude Comparator is no different. Input is provided via slide switches, processed with logic gates, and outputted through LEDs.

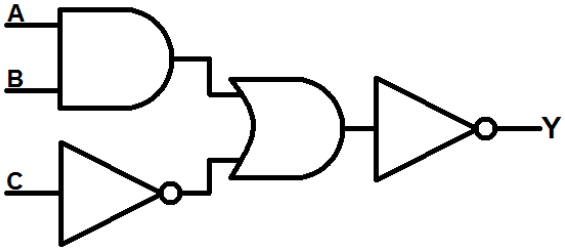



Figure 1. Combinational circuit

There are two different types of logic circuits. The 1-Bit Magnitude Comparator is an example of the first type: *combinational circuits* (see Figure 1). A combinational circuit is a type of digital circuit where the only factor contributing to the output is the current state of the inputs. This means that there is no memory; no part of the circuit can store previous input states. The second type of logic circuit is a *sequential circuit* (see Figure 2). Unlike combinational circuits, sequential circuits have the capability to store the state of inputs. In sequential circuits, the output depends on the current state of the inputs, as well as the past state of the inputs.

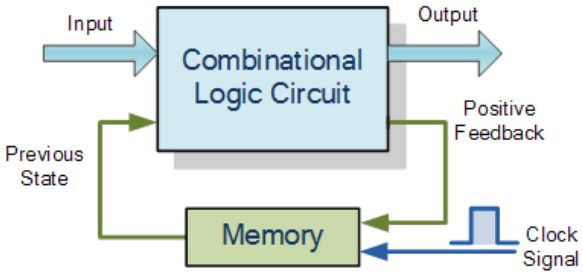



Figure 2. Sequential circuit

## Procedure

The 1-Bit Magnitude Comparator makes use of transistors contained in three different types of integrated circuits (ICs): NOT, AND, and NOR gates. Each gate processes inputs differently; NOT gates are unary operators, taking one input and inverting it (see Figure 3). AND gates are binary operators, taking two inputs, only rendering a high output if both inputs are high. NOR gates are binary operators, taking two inputs, delivering a high output if both inputs are low (see Figure 4). The 1-Bit Magnitude Comparator has two inputs and three outputs (see Figure 1). As it compares magnitudes, the outputs indicate whether input A is greater than, equal to, or less than input B.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | 1 KΩ Fixed Resistor |
| 1 | 9 V Battery |
| 1 | Common Cathode RGB LED |
| 2 | SPDT Slide Switch |
| 1 | Breadboard |
| 1 | CMOS Logic 4069 NOT IC |
| 1 | CMOS Logic 4081 AND IC |
| 1 | CMOS Logic 4001 NOR IC |
| ~ | Wires |

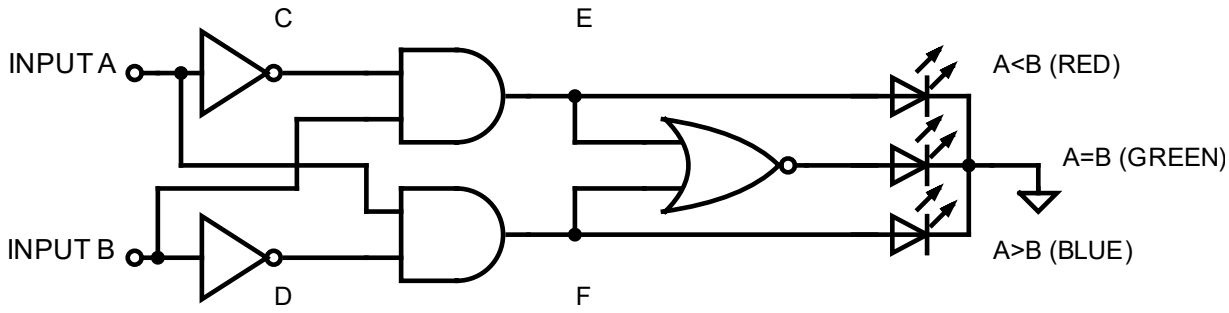



Figure 1. 1-Bit Magnitude Comparator

The 1-Bit magnitude comparator consists of ICs, slide switches (inputs), and an RGB LED (output). Only one output can be active at one time, as the expressions $A < B$, $A = B$, and $A > B$ are mutually exclusive; only one can be true at any given time. A is less than B in exactly one case, and greater than B in exactly one case; when A is high and B is low, A is greater than B. The opposite is true when A is low and B is high.

| A | B | C | D | A<B | A=B | A>B |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |

Figure 2. State of Inputs and Outputs

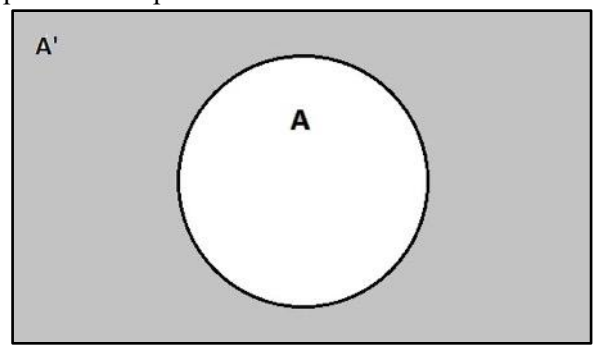



Figure 3. NOT gate

Each binary operator in the circuit is responsible for one output. Due to the inclusion of a NOT gate (see Figure 3), when input A is in its high state and input B is in its low state, points C and D become low and high, respectively (see Figures 1 and 2). This causes the AND gate with inputs at point D (high) and input A (high) to return a high state, signifying that A is greater than B, turning on the blue LED. When input A is low and input B is high, points C and D become high and low, respectively. This causes the second AND gate to return a high state, showing that A is less than B, turning on the red LED. In the other two cases, when A is equal to B, both AND gates return a low state. With both the inputs of the NOR gate in their low state, it returns a high state, signifying that A is equal to B, turning on the green LED.

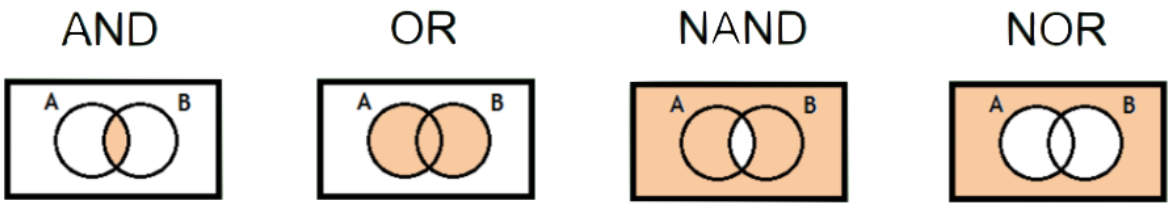



Figure 4. Logic Gates

In order for an input to be in its low state, it must be connected to ground. If it is not connected to anything, it is considered to be in its *floating* state. The floating state is a state that is neither high nor low. It is either somewhere in between, or it changes between high and low based on environmental conditions. If the slide switches are disconnected from the circuit, both inputs are left in the floating state. While input signals fluctuate depending on environmental conditions, the device still functions as a comparator; if both floating signals register as the same input, the LED turns green, and if the floating signals register as different, the LED turns either red or blue. The floating signals can be influenced by many different factors. Touching an IC chip with hands, putting a cellular phone near, or touching the breadboard are all examples of how the floating signals can be caused to fluctuate.

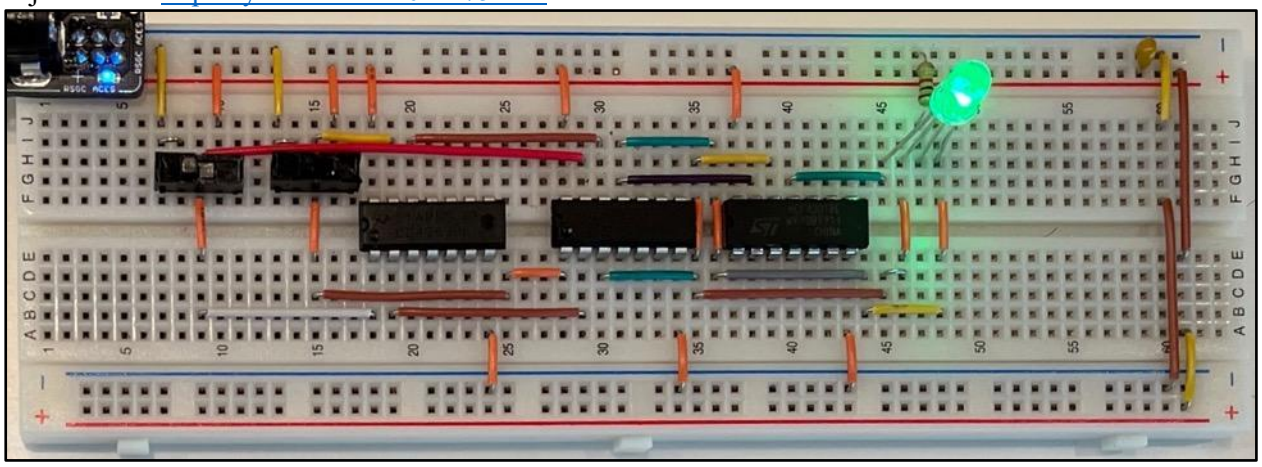
## Finished PCB

The final step in building the 1-Bit Magnitude Comparator is to solder the PCB version. This has a more finished look than the breadboard. This is due to the lack of wires, which are eliminated by traces built into the PCB. These make all necessary connections. The RGB LED as well as the resistor are replaced with their *surface mount device* (SMD) counterparts. SMD components are much smaller than traditional through-hole components and are soldered onto the board with machines when the board is being printed. This makes the PCB even more compact and easy to solder. The only components that need to be soldered on are the IC sockets, switches, and terminal block. The final PCB is then screwed into a 3D printed case, designed by former ACE LC ('21). Metal screw threads are pressed into the case using a hot solder tip, and the PCB is screwed in with black plastic screws.

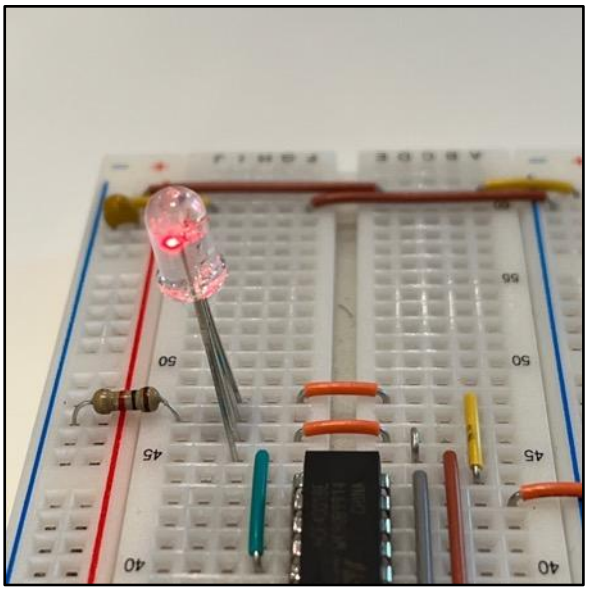
| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | 9 V Battery |
| 3 | 14-Pin IC sockets |
| 1 | Terminal block |
| 2 | SPDT Slide Switch |
| 1 | RSGC ACES 1-Bit Magnitude Comparator PCB |
| 1 | RSGC ACES 3D Printed Case |
| 4 | Metal screw threads |
| 4 | Black plastic screws |
| 1 | CMOS Logic 4069 NOT IC |
| 1 | CMOS Logic 4081 AND IC |
| 1 | CMOS Logic 4001 NOR IC |
| ~ | Solder |

## Media

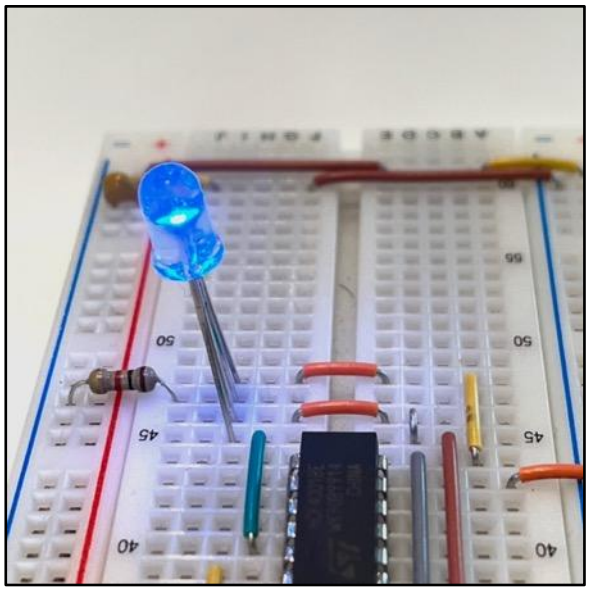
Project video: https://youtu.be/DUJ0KN75eRA



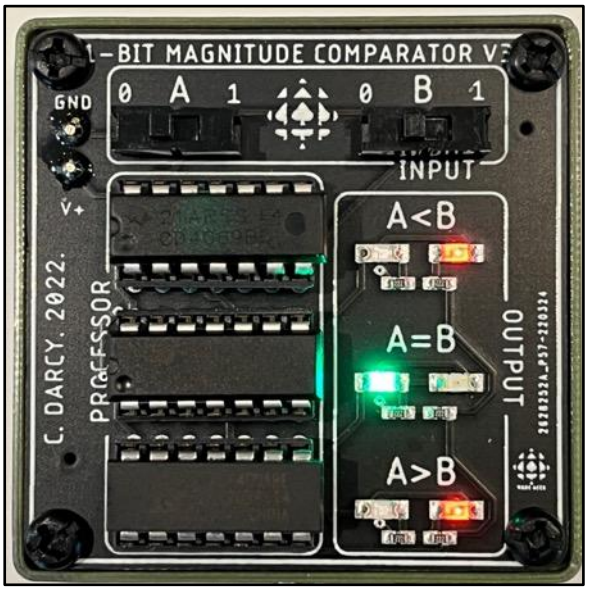
Entire Circuit, A = B (Breadboard Version)



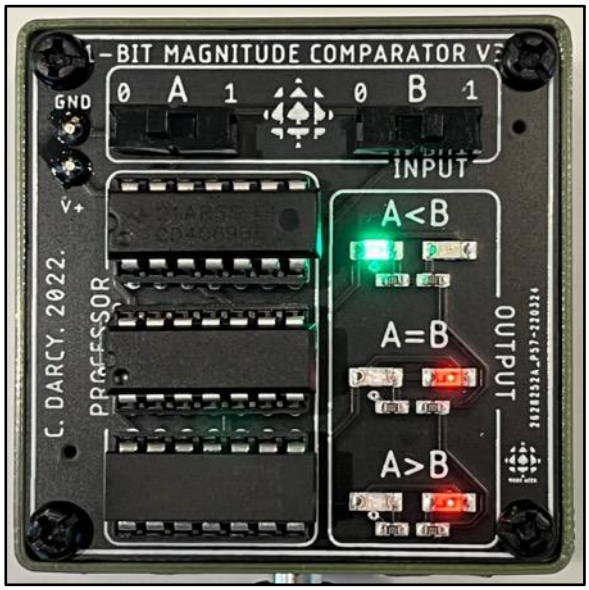
A < B (Breadboard Version)



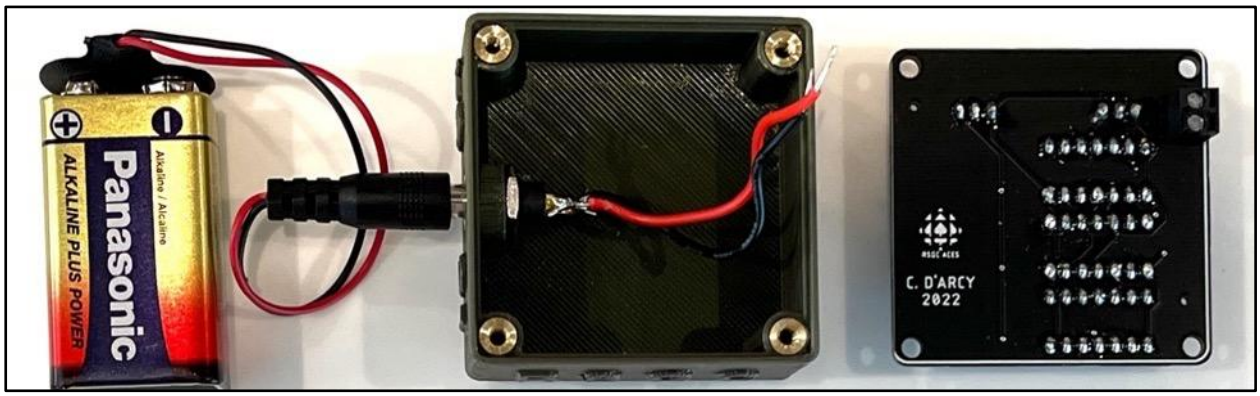
A > B (Breadboard Version



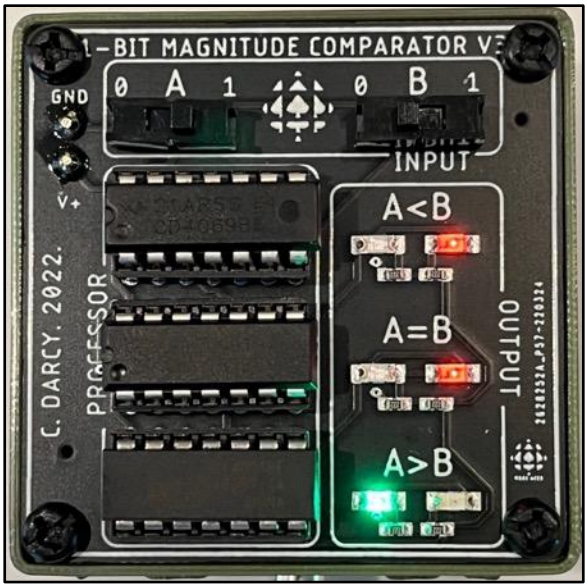

A = B (PCB Version)




A < B (PCB Version)




Battery, Case, and Bottom of PCB




A > B (PCB Version)


PCB Case

## Reflection

This project has definitely been the most interesting for me. Many things have gone right, but many things have also gone wrong. One example of something that did not go according to plan was my attempt to build a functional 2-Bit Magnitude Comparator. A fellow ACE and I were challenged to build a 2-Bit Magnitude Comparator. Thinking it would not be super difficult, we quickly accepted the challenge. After spending many hours painstakingly stripping custom-length wires, wiring up connections, and thinking about which gates we needed to use, the device did not work. Somewhere along the way, we wired an incorrect connection. As much as we tried to debug the comparator, we could not figure it out. This is because we did not plan out our circuit in advance; we just started connecting components as we went along, making the debugging process nearly impossible. While we did not emerge from the endeavor with a 2-Bit Magnitude Comparator, we learned an extremely valuable lesson; planning is not optional. Had we planned out the project in advance, we would have been able to easily debug the system, or better yet, not had any problems to begin with. This is something that we will both remember and ensure to apply to future projects. Going back to the 1-Bit Magnitude Comparator, time management was not an issue. I was able to finish the project before 6 P.M on Saturday. Another challenge that I faced was a lack of parts. I attempted to make a perfboard version of the circuit, however, I did not have enough slide switches, so I had to abandon the project. I think that the 1-Bit Magnitude Comparator was the perfect circuit to use as an introduction to digital logic. This is because it is in that perfect sweet spot where it is complex enough that it requires thinking, but not too complex that it's too ambitious of a first project. Overall, I am pretty happy with this project, and I think it was a success.

# Project 1.4. A Counting Circuit

## Purpose

In addition to exhibiting the distinct functions of some of the many different specialized CMOS 4000 series chips, the purpose of the Counting Circuit is to repeatedly count from 0-9 on a seven-segment display.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#counting

Project schematic: https://crcit.net/c/0f58d7430af64ad9892fa33b57920355

https://www.symmetryelectronics.com/blog/what-are-clock-signals-in-digital-circuits-and-how-are-they-produced-symmetry-blog/

https://learn.circuitverse.org/docs/seq-ssi/clock-signals.html

https://www.rapidtables.com/convert/number/binary-to-decimal.html

https://www.electronics-tutorials.ws/combination/comb_5.html

## Theory

The Counting Circuit (see Figure 1) is a sequential circuit that counts up and down from 0-9 on a seven-segment display. One momentary PBNO acts as the main input for the system. When pressed, a capacitor is instantly charged up. It slowly drains through a large series resistor. As long as the capacitor remains charged above half, the circuit begins to count, with its direction depending on the state of a single pull double throw slide switch. The charged capacitor allows the NAND Gate Oscillator to start generating the clock signal, which is what causes the counting to begin. The clock signal consists of pulses, alternating between high and low, which have both a duration and a frequency. The duration is how long the clock signal runs for, or how long the first capacitor remains charged above half. In this case, that time is approximately 7.5 seconds (see section B). The frequency is the number of cycles completed per second, measured in hertz. The frequency of the clock signal is 7 Hz, as measured with a stopwatch. The clock signal gets passed into the Decade Counter, which repeatedly counts from 0-9 on each pulse. Instead of using the 0-9 output of the decade counter, the divide 10 pin, which is high from 0-4 and low from 5-9, is used. This effectively divides the frequency of the clock signal by 10. This new clock signal is fed into another type of decade counter; this decade counter counts from 0-9 in binary. The four outputs of this chip (1, 2, 4, 8) are connected to one final chip: a binary counting decimal decoder. This chip converts the presented binary number to the correct combination of LEDs on a seven-segment display. The result is a repeated counting sequence from 0-9.

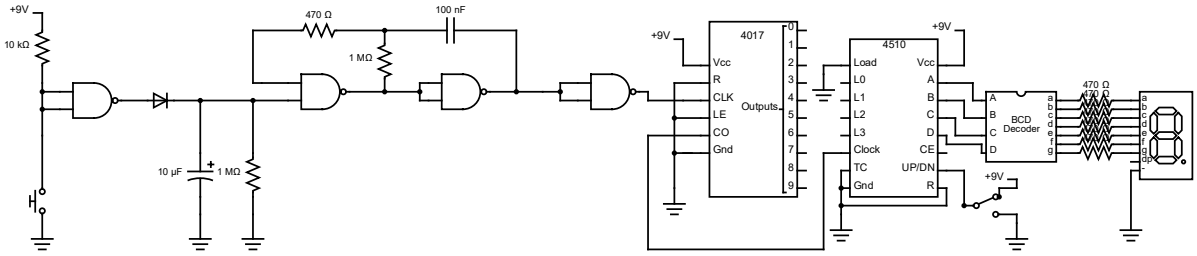



Figure 1. A Counting Circuit

# A. Analog Input

## Purpose

The purpose of the Analog Input is to charge up a capacitor in the NAND Gate Oscillator, starting the count of the Counting Circuit.

## Reference

Section description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#CCA

Section schematic: https://crcit.net/c/4fac7886e4f0430c96fc78c14cb29f34

## Procedure

The purpose of the analog input section of the Counting Circuit is to control when the circuit counts; it contains a momentary PBNO, that when pressed, starts the clock signal. The Analog Input section of the circuit uses *low side switching* (see Figure 1). Low side switching is when the output is normally high, and it relies on a *pull-up resistor*. A pull-up resistor is used to ensure that the output remains in a high state when no input is present.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | PBNO |
| 1 | 10 KΩ Fixed Resistor |

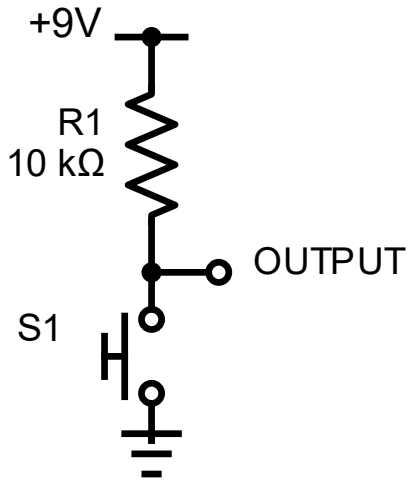
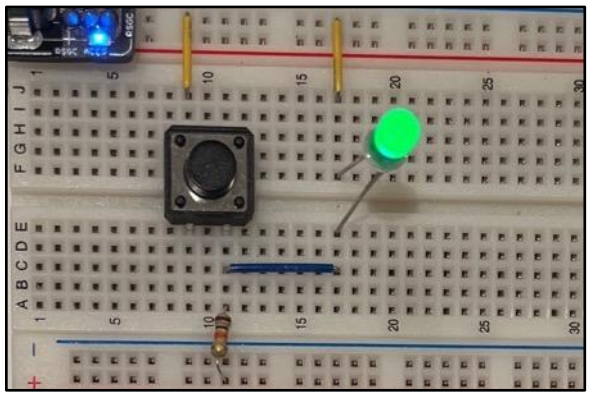


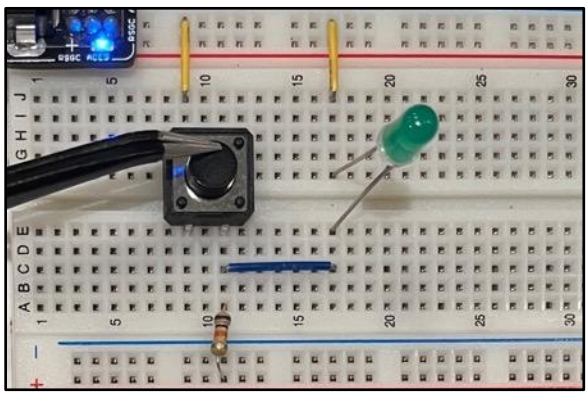

Figure 1. Analog Input

At rest, the output is connected to the supply voltage, registering a high signal; as soon as the button is pressed, however, the output is connected directly to ground, without any resistance. This causes the output to register as low. If the output were connected to a logic gate, the full, unimpeded 9 V could be presented, instead of going through a resistor.

## Media



PBNO Open (with LED for Visualization)



PBNO Closed (with LED for Visualization)

## Reflection

When I first built this section, I didn't think that it deserved its own section. I thought it should have been included with the NAND Gate Oscillator. After writing about, I have realized that it is actually one of the most important sections; even though it doesn't have the most parts, there is a lot of theory behind how it works. This is one of things I love about engineering: the most complex and important things are often the most subtle.

## B. NAND Gate Oscillator

### Purpose

The purpose of the NAND Gate Oscillator (NGO) is to provide a clock signal for the entire Counting Circuit, acting as a metronome and allowing it to perform a sequence of actions in time.

### Reference

NAND gate datasheet: http://darcy.rsgc.on.ca/ACES/Datasheets/CD4011BC.pdf

Section description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#CCB

Section schematic: https://crcit.net/c/2345f608f48a467b977aae5a9d244656

https://byjus.com/question-answer/which-of-the-following-is-the-universal-gate-or-gate-and-gate-nand-gate-not/

https://www.electronicshub.org/sequential-circuits-basics/

### Procedure

The NAND Gate Oscillator (NGO), like any machine, contains three sections: input, processing, and output. The final output section renders a *square wave*. A square wave is a signal that switches between high and low states with equal amounts of time in each state. The NAND Gate Oscillator accomplishes this by using NAND gates in conjunction with two resistor-capacitor pairings.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | Analog Input (See Part A) |
| 2 | 1 MΩ Fixed Resistor |
| 1 | 470 Ω Fixed Resistor |
| 1 | 10 µF Electrolytic Capacitor |
| 1 | 0.1 µF Disk Capacitor |
| 1 | 5mm LED |
| 1 | Signal Diode |
| 1 | CMOS Logic 4011 NAND IC |
| 1 | 9V Power Source (Battery) |
| 1 | Breadboard |
| ~ | Wires |

The CMOS 4011 IC is a 14-pin dual in-line package chip (DIP) containing the NAND gates. NAND gates are the inverse operator of AND gates, rendering a high output in every case except for when both inputs are high. The function of this chip is to present either supply voltage or a path to ground on its output pins when certain input conditions are met. When used repeatedly, NAND, also known as the universal gate, can emulate other logic gates. For example, by wiring both inputs together, a NAND gate functions as a NOT gate. NAND gates can emulate many operators including AND, OR, NOR and XOR (see Figure 1).

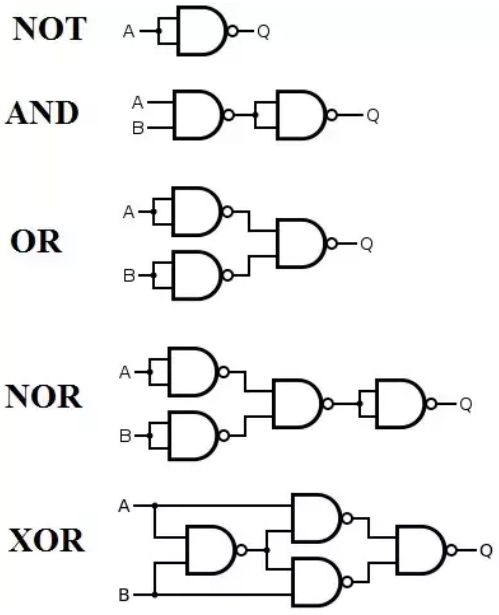



Figure 1. NAND: The Universal Gate

NAND gates take the input signal, process it, and output a value corresponding to the inputs. There are a total of four possible combinations of inputs and outputs. All of them are high except for when both inputs are high. In this case, the output is low. The 4011 chip is able to take an analog (continuous) input and give a digital (discrete) output. The 4011 IC can take any input voltage between 3 V and 15 V, but only outputs a value of 0 or 1 (see Figure 4).

If the voltage on the input pin is less than half of the supply voltage, it outputs a low signal, or ground. If the voltage on the input pin is greater than half of the supply voltage, it outputs a high signal, or supply voltage. The IC contains a total of 14 pins and four NAND gates. Each gate requires three pins: input A, input B, and output. The inputs and outputs of the four NAND gates occupy 12 of the 14 total pins. The output pins are positioned between one set of input pins on either side. The remaining two pins are used for ground and power. Like most electronic devices, the IC cannot function without being connected to power; in addition to having a voltage applied on the inputs, a supply voltage must be present on pin 14 (see Figure 2). This is where any electrical energy on the output pins come from.

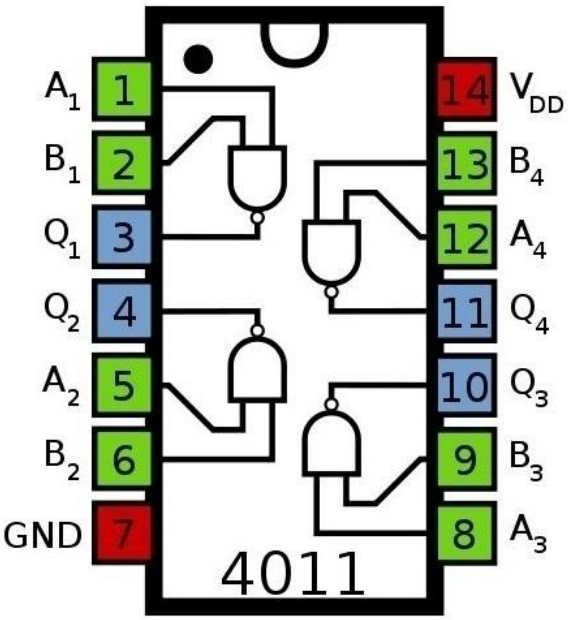



Figure 2. CMOS 4011 IC Pinout

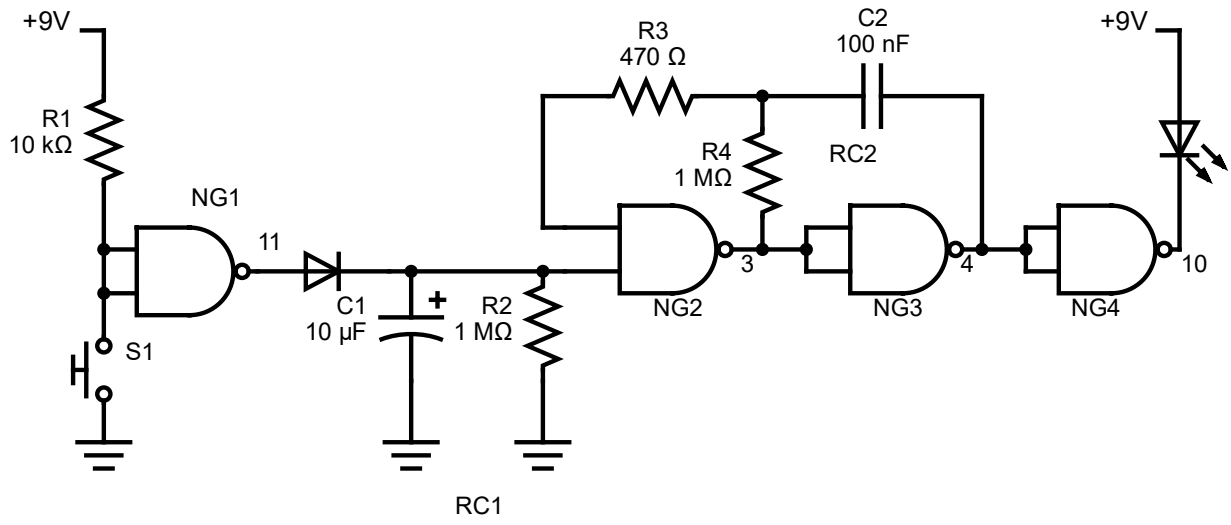



Figure 3. The NAND Gate Oscillator

The Counting Circuit is a sequential circuit, meaning that in addition to the current state of the input, its outputs depend on the past states of the input. In order for a sequential circuit to change its memory elements, it needs a *clock signal*. A clock signal is a synchronized signal that oscillates between high and low states at a constant rate. In the case of the counting circuit, the clock signal is in the form of a square wave, generated by the NAND Gate Oscillator (see Figure 3). Electrical components and logic gates have no understanding of time without a clock signal, therefore, if a sequential circuit such as the Counting Circuit did not have a clock signal, the output of the circuit would not change at all.

The NAND Gate Oscillator incorporates two resistor-capacitor pairings (see Figure 3). The first of which, RC1, controls how long the NGO, and by extension, the Counting Circuit, operates; in order for the Counting Circuit to change its output, the clock signal generated by the NGO must be running.

At rest, both inputs of NG1 are high, meaning that its output is low, and C1 remains discharged. This means that at least one input of NG2 is low and its output is high. In their current configuration, NG3 and NG4 act as NOT gates (see Figure 1) meaning that the output of NG4 is high. Since both sides of the LED are connected to power, it does not light up.

When the button pressed, NG1 outputs a high signal on pin 11. Since the capacitor is connected to pin 11 through a diode, it charges up almost instantly. A diode has a voltage drop of 0.7 V, meaning that the capacitor is charged to 8.3 V. Even when the button is released, the capacitor does not drain through NG1 since the diode inhibits current from flowing in that direction. The only path that the capacitor has to drain is through the R2.

The amount of time that a capacitor takes to discharge is 5T where tau is represented by the equation $T = RC$ (see Figure 5). In this case, T is equal to $10\ \mu F \times 1\ M\Omega = 10$, giving the capacitor a total discharge time of $10 \times 5 = 50$ seconds.

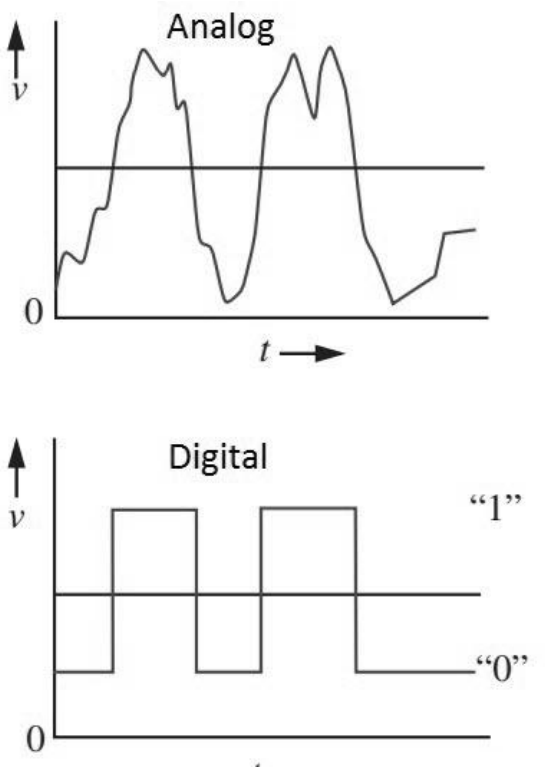



Figure 4. Analog to Digital Conversion

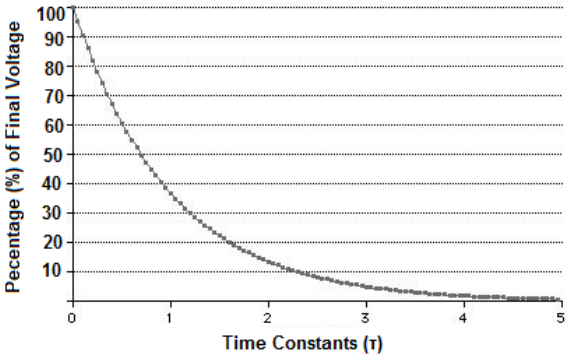
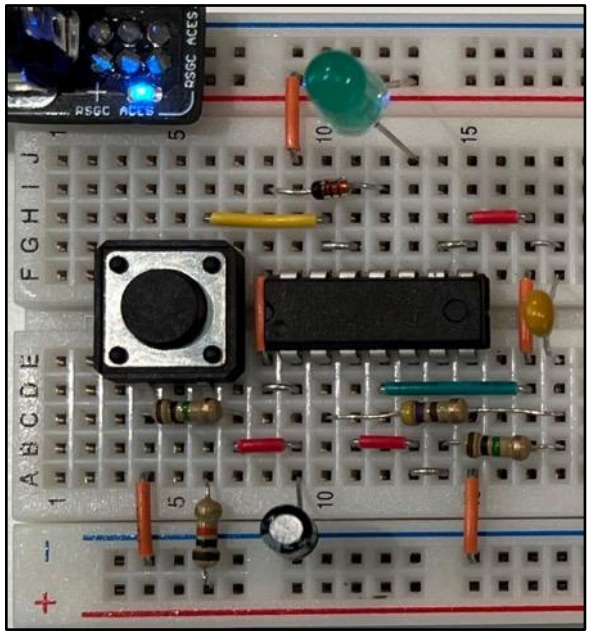



Figure 5. Capacitor Discharge Curve

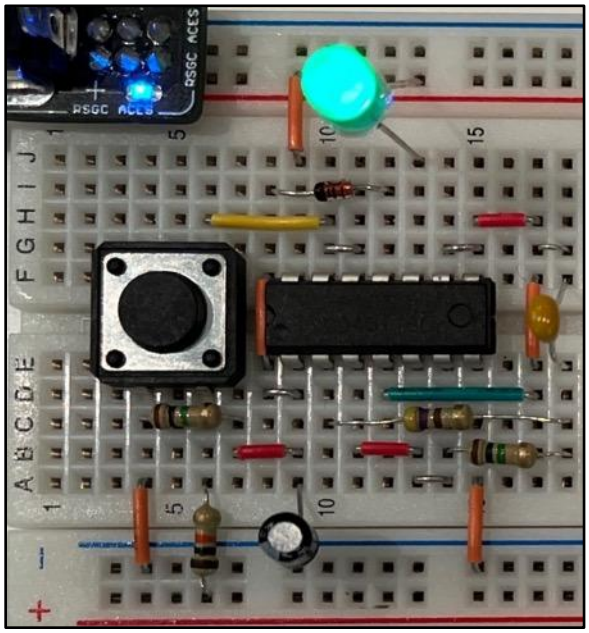
If the discharge curve of a capacitor were linear, the NGO would run for precisely half of the discharge time, when the voltage remains above 4.5 V, or 25 seconds; the curve, however, is not linear. The capacitor reaches a charge value of 50 percent after approximately 0.75T, or 7.5 seconds (see Figure 5). This means that the NGO runs for approximately 7.5 seconds after the button is released. If either the resistor or capacitor were increased or decreased, the NGO would stay on longer or shorter, respectively,

RC2 is responsible for generating the square wave. When RC1 is charged, a high signal on NG2 goes through R4 and fills up C2 until the resistance is more than 470 Ω. The signal then feeds back on itself into NG2's input. If RC1 remains charged, NG2 receives two high signals and outputs low. This causes C2 to drain and NG3 to output a high signal. NG4 acts as a NOT gate, so it outputs a low signal. When the capacitor no longer provides half the input voltage, one of the inputs of NG2 registers a low signal and it outputs high again. This process repeats constantly until C1 drains below 4.5 V.

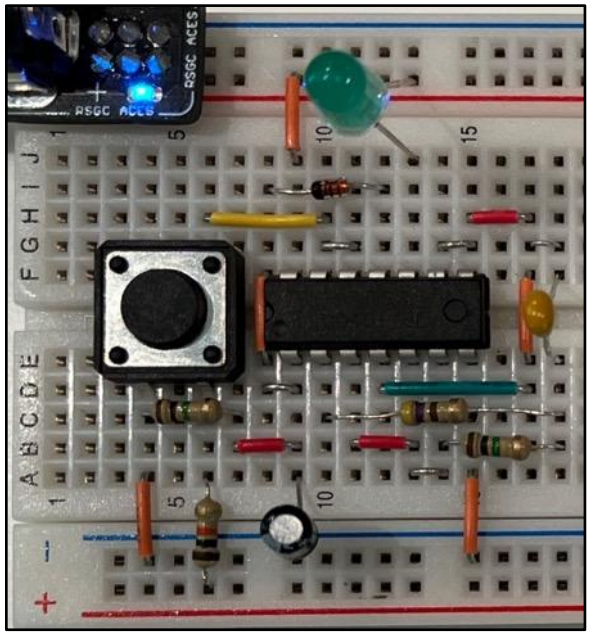
## Media



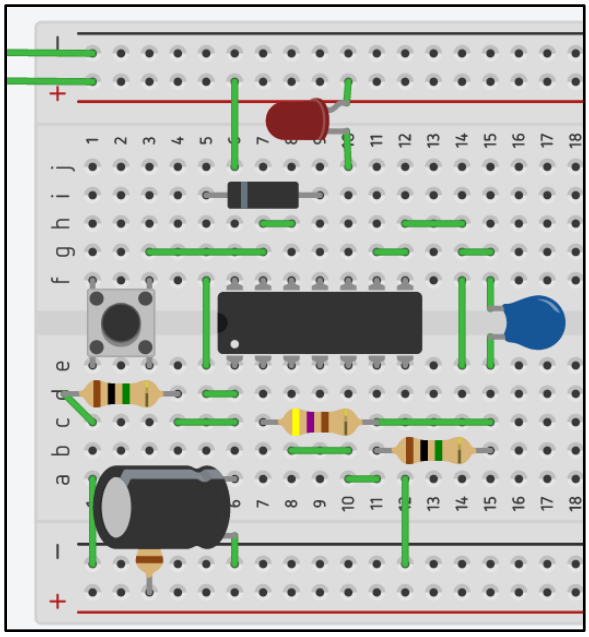

C1 Discharged




C1 Charged (Mid Oscillation, NG4 low)




C1 Charged (Mid Oscillation, NG4 high)



TinkerCAD Diagram

## Reflection

This part of the counting circuit took some time to get working; I initially used the wrong capacitor in the second RC circuit. Instead of the 104, I used a 103. While this does seem like a bad thing, it ended up being helpful in the long run. When I used the 103, since it had 10X less capacitance, the LED flickered extremely quickly. This helped me understand how the NGO works. I think that this is the value of a hands-on course; I got to see the resistor-capacitor pairings in action. Overall, this section of the report has taught me the importance of the clock signal and how crucial NAND gates are in modern electronics. With my understanding of the NGO, I am now ready to build and write about the next section.

## C. Decade Counter

### Purpose

The purpose of the Decade Counter is to divide the frequency of the clock signal, provided by the NAND Gate Oscillator, by 10. This allows the circuit to count slower, at the desired rate.

### Reference

Decade counter datasheet: http://darcy.rsgc.on.ca/ACES/Datasheets/cd4017b.pdf

Section description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#CCC

Section schematic: https://crcit.net/c/71ac9626255a43efaf30e9f312d55aff

https://www.electronicshub.org/ic-4017-decade-counter/

### Procedure

The Decade Counter (4017) is a 16-pin DIP logic IC. It is a synchronous counter, meaning that it requires a clock signal input. It is provided by the NAND Gate Oscillator (see part A). The Decade Counter repeatedly counts from 0-9 while the clock signal continues to run. The count advances on the high pulse of the clock signal and lasts one full clock signal cycle. Each output pin corresponds to a number between 0 and 9 (see Figure 1), outputting the supply voltage on each count (see Figure 2). The final output shown, labelled "OUT," corresponds to the "÷10" pin shown in Figure 1, and is often referred to as carryout. For the counts from 0-4, carryout is high. For counts 5-9, it is low. This effectively divides the frequency of the clock signal by 10; every 10 cycles of the clock signal leads to one cycle on the carryout pin.

This function is especially useful when two frequencies are desired: for example, one Decade Counter's clock input could be connected to an NGO, and a second Decade Counter could have its clock input connected to the carryout of the first one. This would create a device that could count to 99 instead of 9.

In order for the Decade Counter to function properly, each input pin must be properly *conditioned*. This means that the pins should be directly connected to power or ground, depending on the desired function, and the type of pin. Pins can either be *active high*, or *active low*. Active high means that the function is obtained with a high signal. Active low means the opposite.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | NGO (See Part B) |
| 1 | CMOS 4017 Decade Counter IC |
| 1 | 10 Segment LED Bar |
| 1 | 9V Power Source (Battery) |
| 1 | Breadboard |
| ~ | Wires |

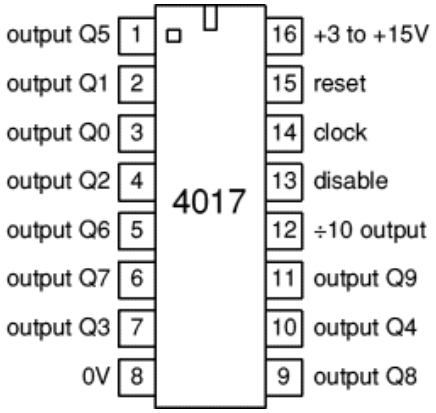



Figure 1. Decade Counter Pinout

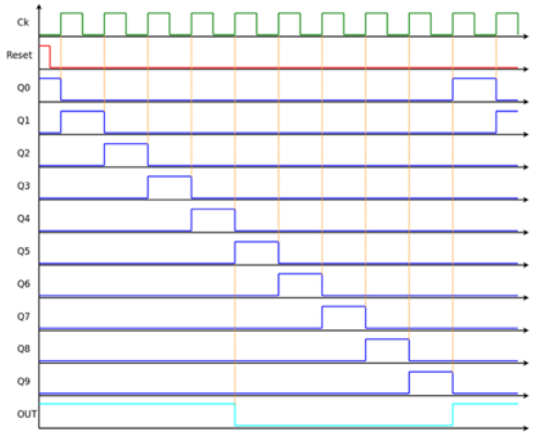



Figure 2. Decade Counter Signal Output

All the input pins on the Decade Counter are active high. This includes both the reset and disable pin. In order to properly condition these pins so that chip counts properly, both must be grounded. When reset is high, the count is reset back to 0. When the disable pin is high, the entire chip is disabled, ignoring the clock signal.

The reset pin serves dual functions: A reset button could be created with a high-side switch or button between reset and the supply voltage (see Figure 3). This would make use of a pull-down resistor, resetting the count to 0 when pressed. In this configuration, at rest, the pull-down resistor connects the pin to ground; when the button is pressed, however, the reset pin is connected to power, without resistance, activating it, returning the count to 0. The same could be done with the disable pin, creating a stop/start button/switch.

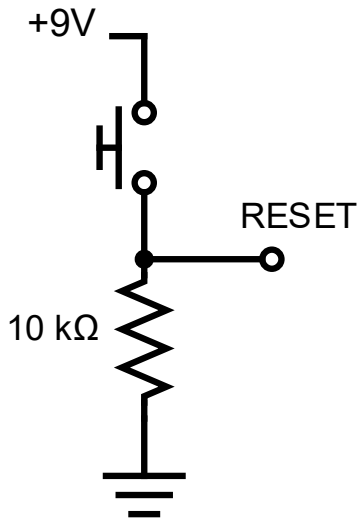



Figure 3. Decade Counter With Reset

The reset pin could also be used to bypass outputs above a certain count. For example, to count from 0-7, repeatedly, on the clock pulse, output Q8 would have to be connected to reset. This function of the reset pin allows for the creation of uneven square wave outputs; the carryout pin is also affected by this change. Since it is high from 0-4, and low 5-9, when only counting to 7, it is high for 5 counts, but only low for 3. This effectively creates a variability in division of the carryout pin; since the Decade Counter only performs actions on the high pulse of the clock signal, the clock signal frequency is divided by 8.

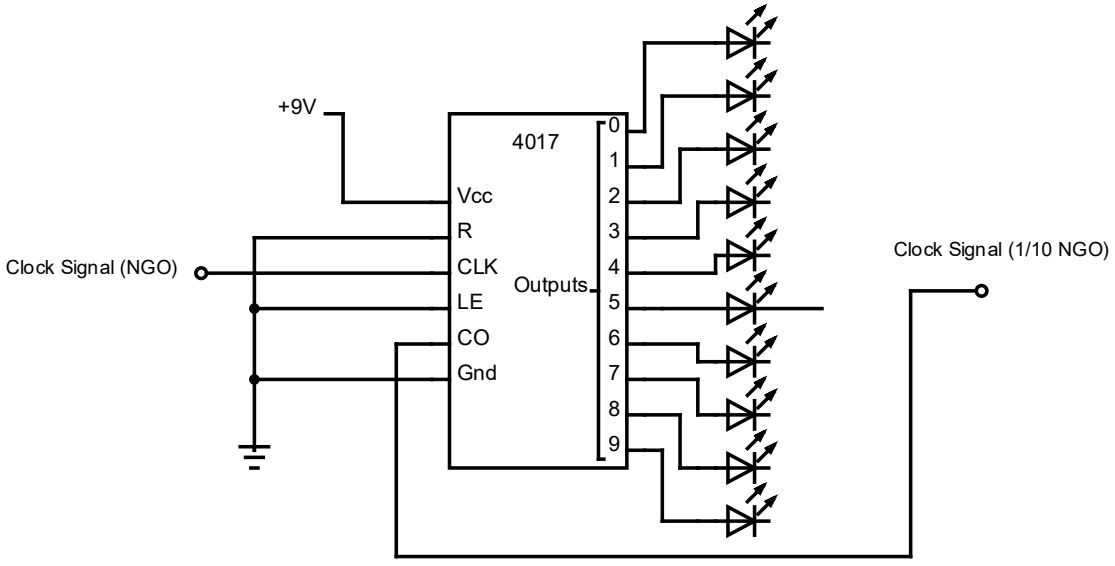
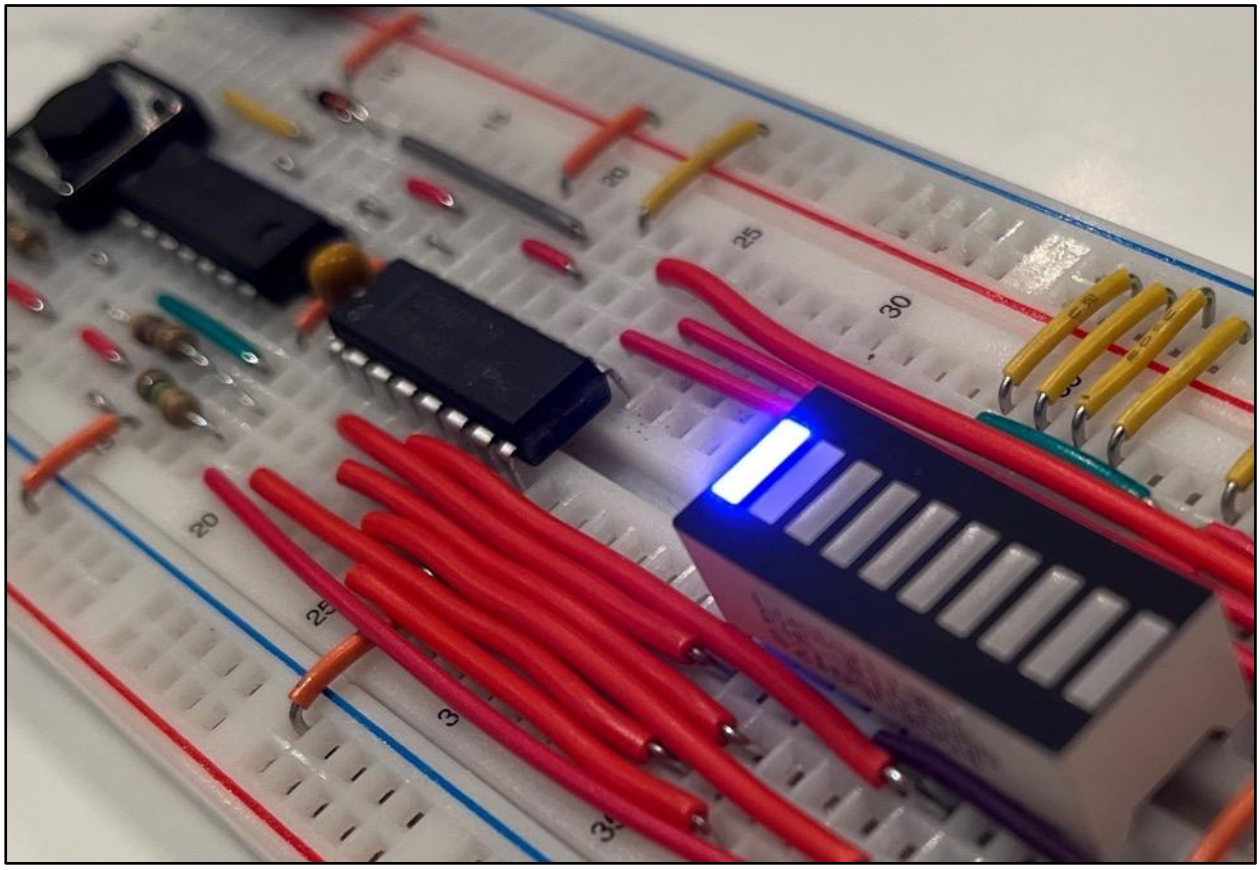


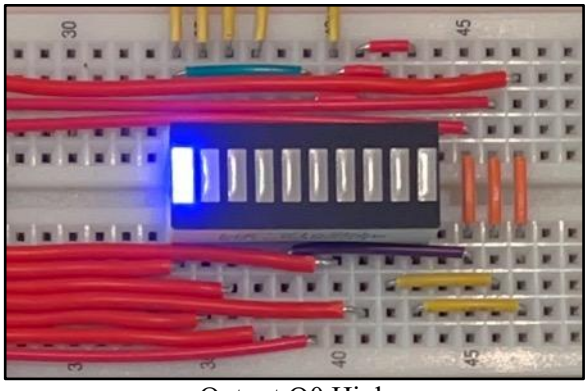

Figure 4. Decade Counter

The Decade Counter chip can be setup with 10 LEDs, so that each LED is active for one clock signal cycle (see Figure 4). Instead of individual LEDs, each output can be connected to one segment of an LED bar. This creates an animation, with the light "sweeping" across the bar. The left side of Figure 4 shows the appropriate connections that must be made to properly condition the chip.

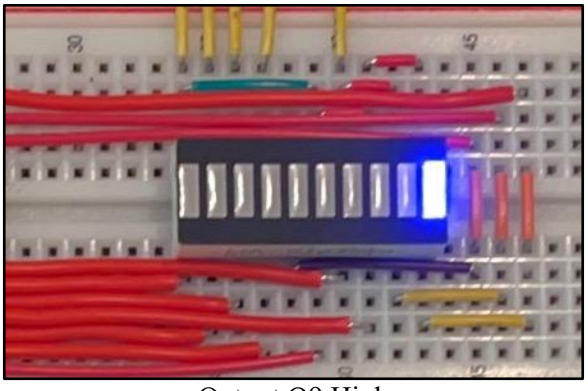
## Media



The Decade Counter



Output Q0 High



Output Q9 High

## Reflection

This section of the report was extremely helpful for me; I know that we could have just thrown the Decade Counter IC into the circuit, wired up the clock and carryout, and called it a day, but building the circuit with the LED bar was a valuable step. By building the bar, I was taught a number of things: the way the Decade Counter divides the clock signal by 10, but more importantly, I was taught the value of presentation. I was taught how even though jumper wires would allow the circuit to be functionally identical, it would be a completely different circuit. This is why I decided to cut my own custom wires; in a report, form is as important as function. Overall, I now understand the function of the Decade Counter, and I am ready to build and write about the Decimal Binary Counter.

## D. Decimal Counting Binary Up/Down Counter

### Purpose

The purpose of the Decimal Counting Binary Up/Down Counter (DCBUC) is to count from 0-9 in both directions, in binary, on each pulse of the divided clock signal provided by the Decade Counter.

### Reference

Decimal counting binary counter datasheet: http://darcy.rsgc.on.ca/ACES/Datasheets/cd4516b.pdf

Section description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#CCD

Section schematic: https://crcit.net/c/59d2c23261de4695ad34a16fbd78edfc

https://www.build-electronic-circuits.com/4000-series-integrated-circuits/ic-4510/

### Procedure

Like the Decade Counter, the Decimal Counting Binary Up/Down Counter (4510) is a synchronous 16-pin DIP logic IC. The clock signal comes from the carryout pin on the Decade Counter. On each rising edge of the clock signal, the count advances. This result is a binary counter that counts at a rate 10 time slower than the NAND Gate Oscillator (see part A).

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | Decade Counter (See Part C) |
| 1 | CMOS 4510 IC |
| 1 | 9V Power Source (Battery) |
| 1 | Breadboard |
| ~ | Wires |

The DCBUC contains three conditioned pins for normal operation (see Figures 1 and 2): reset, preset, and carry in must be connected to ground, as they are active high pins and their functions are not desired when operating normally. In Figure 2, the load pin corresponds to the preset pin. The up/down pin is low for backwards counting, and high for forwards counting. The line above down shown in Figure 1 means that the down function is active low.

When high, the preset pin is used to set the number that the counter starts from. This is also what inputs A-D are used for; to store a number, the desired number is presented in binary on inputs A-D, and the preset pin is briefly connected to the supply voltage. After this, when the count is reset, instead of being reset to 0, it is reset to the preset number.

In the way the DCBUC is configured in the Counting Circuit (see Figure 2), the preset function is not used; they can be left floating because the preset pin is grounding. It does not matter if anything is on the input pins, they are ignored since preset is low.

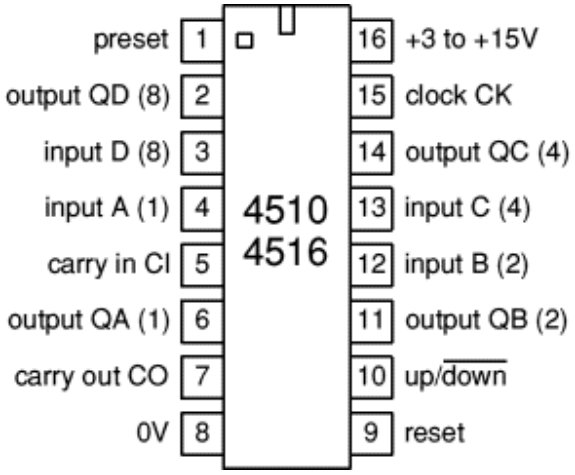



Figure 1. Decimal Binary Counter Pinout

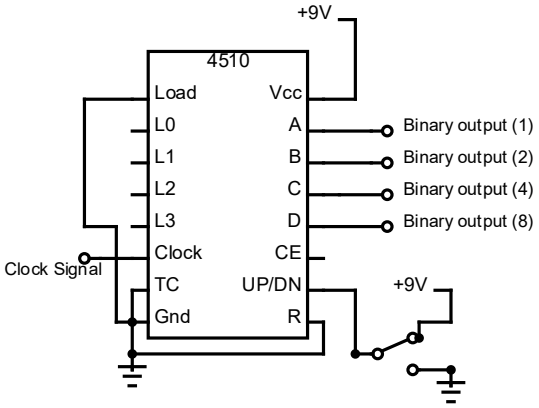



Figure 2. Decimal Binary Counter Connections

| Figure 3.<br>4 Bit Binary Values in Decimal | | |
|---|---|---|
| **Binary Number** | **Bits Visualized** | **Decimal Equivalent** |
| 0000 | 0+0+0+0 | 0 |
| 0001 | 0+0+0+1 | 1 |
| 0010 | 0+0+2+0 | 2 |
| 0011 | 0+0+2+1 | 3 |
| 0100 | 0+4+0+0 | 4 |
| 0101 | 0+4+0+1 | 5 |
| 0110 | 0+4+2+0 | 6 |
| 0111 | 0+4+2+1 | 7 |
| 1000 | 8+0+0+0 | 8 |
| 1001 | 8+0+0+1 | 9 |
| 1010 | 8+0+2+0 | 10 |
| 1011 | 8+0+2+1 | 11 |
| 1100 | 8+4+0+0 | 12 |
| 1101 | 8+4+0+1 | 13 |
| 1110 | 8+4+2+0 | 14 |
| 1111 | 8+4+2+1 | 15 |

Binary, just like decimal, is a number system. Decimal is base 10, meaning that there are 10 distinct digits: 0-9. After 9, a second digit is introduced, which counts to 9 again. In binary, it is the same concept, except it is base 2. Starting from the right, the first digit represents 1s, the second, 2s, the third, 4s, and the fourth, 8s (see Figure 3). Each subsequent digit represents its respective power of 2.

In binary, instead of having 10 distinct digits there are two: 0 and 1; or high and low. This is how the DCBUC counts; a high signal on an output represents a 1, whereas a low signal represents a 0 (see Figure 4). Each output oscillates at a frequency half the magnitude of the last; the clock signal completes two full cycles for each cycle the ones output completes, four cycles for the twos output, eight cycles for the fours output, and so on. This means that the DCBUC could technically count from 0-16 instead of 9; since it is a binary decimal counter, however, it only counts 9 in, binary. The DCBUC has a "sister" chip, the 4516, that has the same function, but counts to 15 instead of 9.

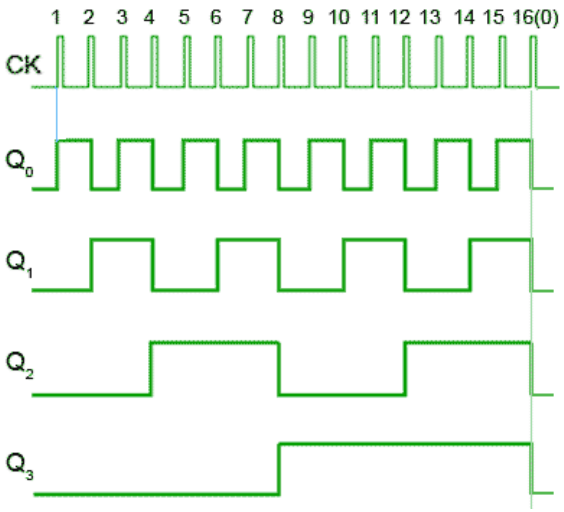
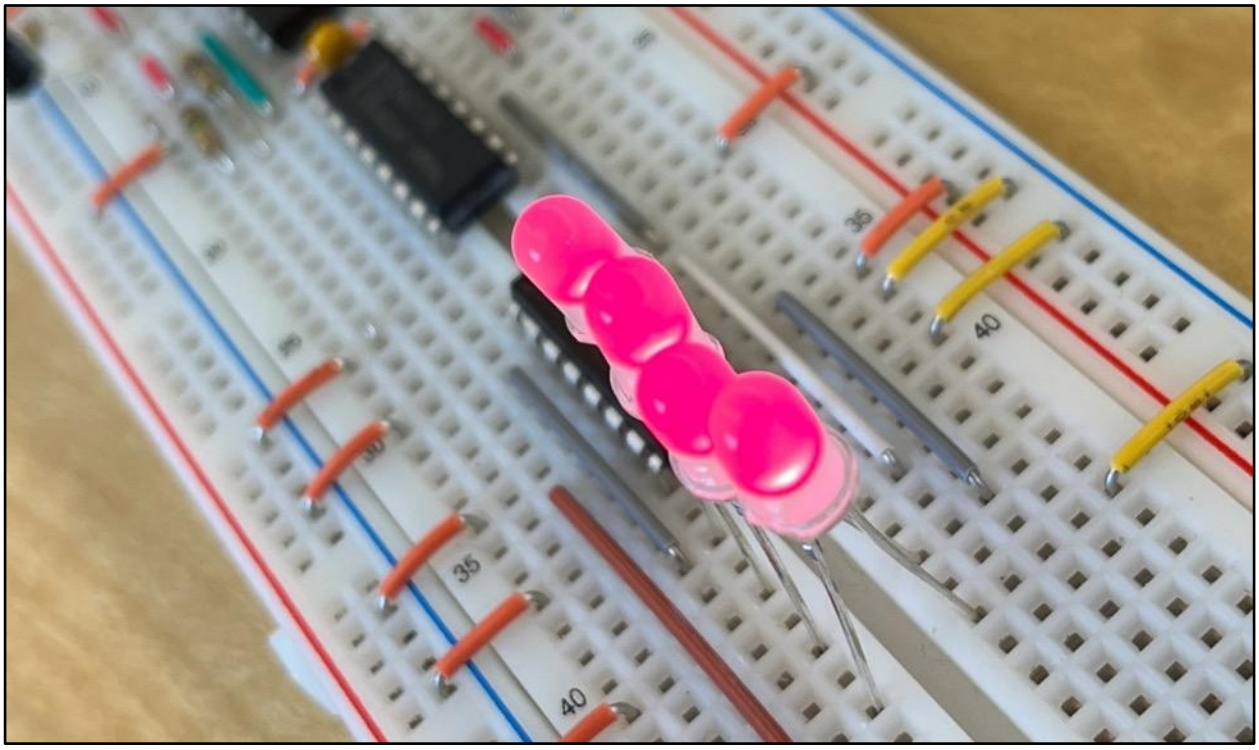


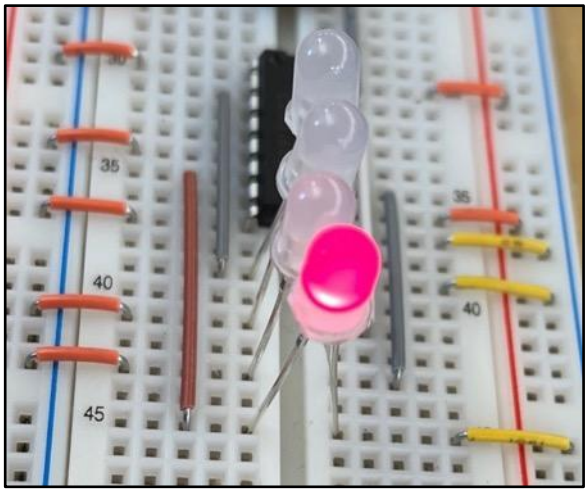

Figure 4. Binary Counter Output Signal (0-16)

This is the binary equivalent to the carryout pin on the Decade Counter (see part C). Just like the Decade Counter carryout pin divides the clock signal by 10, output pins A-D on the DCBUC can be thought of as dividing the clock signal by 2, 4, 8, and 16 respectively. The only difference is that the DCBUC only counts to 9, not 16.

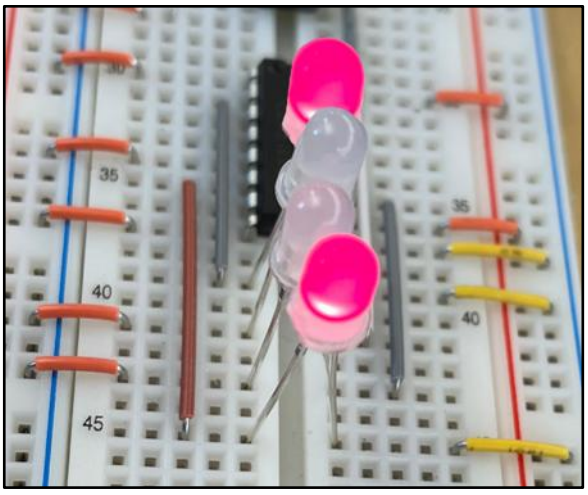
## Media




Counting Binary Up/Down Counter (with 4516)




Binary Count 1




Binary Count 9

## Reflection

When I first started building (and writing about) the DCBUC, I thought to myself, "I'm not going learn anything new, it's literally the same as the Decade Counter, but in binary. I'll just write the section and call it a day." Multiple hours of building and technical writing later, I have proven myself wrong. I have since learned the intricacies of the DCBUC, and how it differs from the Decade Counter. I have learned how similar binary actually is to decimal. I understand that they are different, but after this part of the DER, I have uncovered many similarities. Overall, this section has taught me lessons about the DCBUC, but more importantly, it's taught me about the writing process; I now understand why the Counting Circuit is done in sections instead of one long report.

## E. Binary Counting Decimal Decoder

### Purpose

The purpose of the Binary Counting Decimal Decoder (BCD Decoder) is to convert the binary counting signal provided by the Decimal Counting Binary Up/Down Counter (see part D) to a signal that can correctly control the segments of a seven-segment display.

### Reference

Binary counting decimal decoder datasheet: http://darcy.rsgc.on.ca/ACES/Datasheets/cd4511b.pdf

Section description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#CCE

Section schematic: https://crcit.net/c/5b1f4e609a244c6c86aaa15eab137fc8

### Procedure

The BCD Decoder is a 16-pin DIP IC. When the store pin is low (see Figure 1), the chip features only combinational logic; this means that the outputs depend solely on the current state of the inputs. Each combination of inputs A-D yields a different combination of outputs on pins a-g, lighting up the correct LEDs to show the inputted number on the seven-segment display (see Figure 3).

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | DCBUC (See Part D) |
| 1 | CMOS 4511 BCD Decoder IC |
| 1 | 9V Power Source (Battery) |
| 1 | Breadboard |
| ~ | Wires |

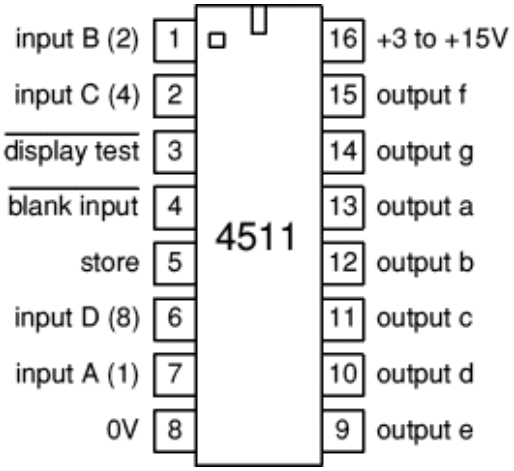



Figure 1. BCD Decoder Pinout

When the store pin is high, the chip switches to sequential logic. The IC internally stores the state of the input pins. As long as the store input remains high, the current inputs are ignored, and outputs a-g remain in the states they would be when the inputs were stored.

The display test and blank input pins perform similar functions to the store pin. The line over these pins in Figure 1 indicates that they are active low. This means that for normal operation, they must be high. When the display test pin is low, outputs a-g are high, ignoring inputs A-D. This is useful when debugging; if a component is not operating correctly, grounding the display test pin can quickly confirm that each segment of the display is functioning. Conversely, when the blank input pin is grounded, outputs a-g are low, deactivating segments a-g (see Figure 2).

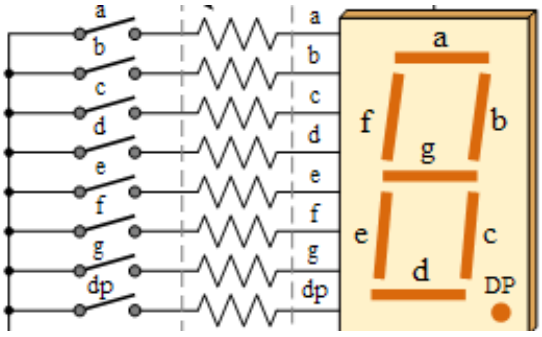



Figure 2. Labelled Seven Segment Display

The BCD Decoder does not change the input data The DCBUC outputs a number, in binary-coded decimal form. The BCD Decoder simply converts the number from BCD form to a seven-segment display signal across seven pins.

**Figure 3.**
**BCD Decoder Truth Table**

| INPUT | | OUTPUT | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Decimal | Binary | A | B | C | D | E | F | G |
| 0 | 0000 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0001 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0010 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 3 | 0011 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 4 | 0100 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 5 | 0101 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 6 | 0110 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 7 | 0111 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 8 | 1000 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1001 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

Inside the BCD Decoder is an arrangement of AND, OR, and NOT gates (see Figure 4). The gates are arranged in a way that signals on inputs A-D cause outputs a-g to output the correct signals to display the given BCD number on a display (see Figures 2-4).

If a BCD Decoder IC was not available, the outputs of the Decimal Counting Binary Up/Down Counter could be connected in the configuration shown in Figure 4. This would serve the exact same purpose; the number inputted in BCD would get converted to seven-segment display-coded outputs a-g.

In Figure 4, each output is isolated and outlined in red, allowing every pin's output to be retraced. In the configuration where logic gates are used in place of a BCD Decoder, this is helpful for debugging. The incorrectly-functioning output could be debugged with the help of Figure 4. The isolated outputs would greatly simplify the debugging process, allowing one output to be debugged at a time.

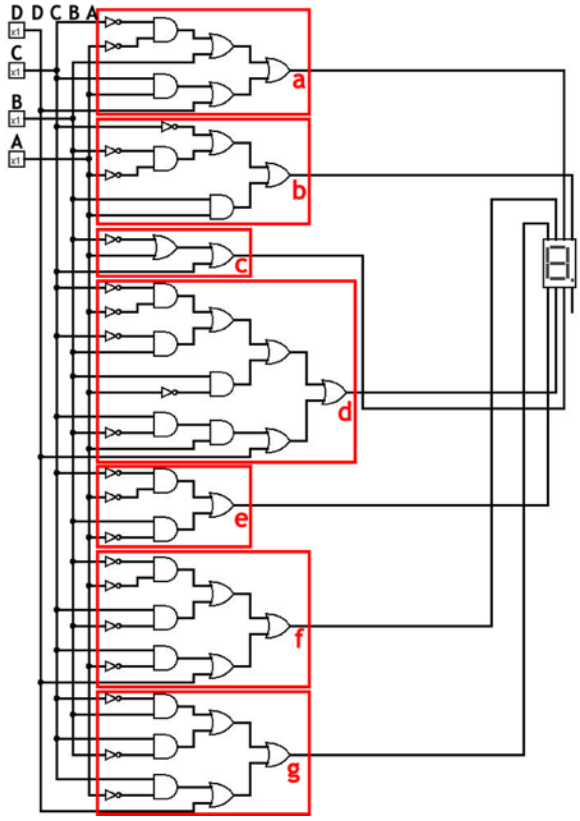
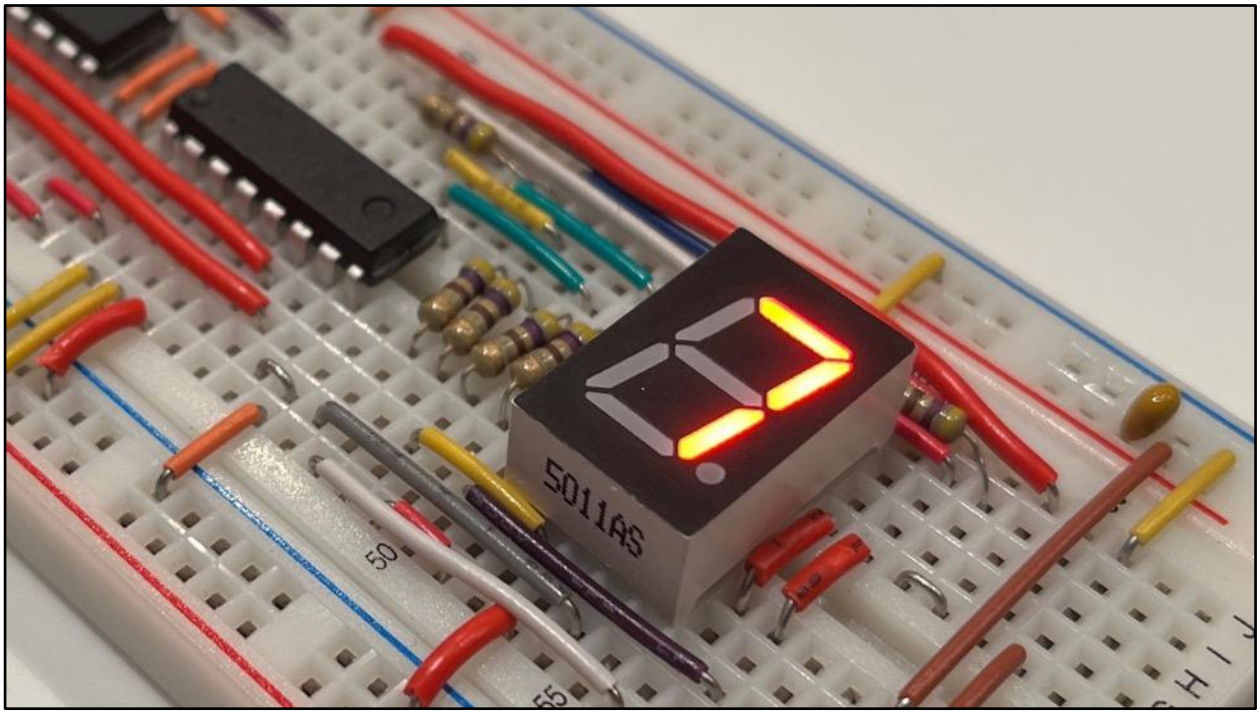


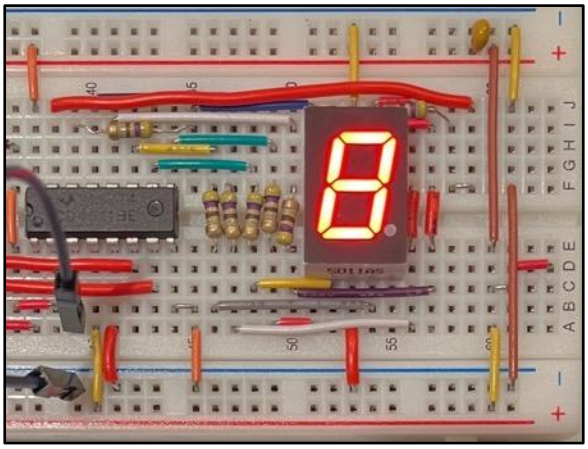

Figure 4. Inside the BCD Decoder

The only difference in this configuration is that there are no special functions; the circuit would only act as a combinational decoder. There would be no store, display test, or blank input functions. The outputs would depend only on the current state of the inputs.

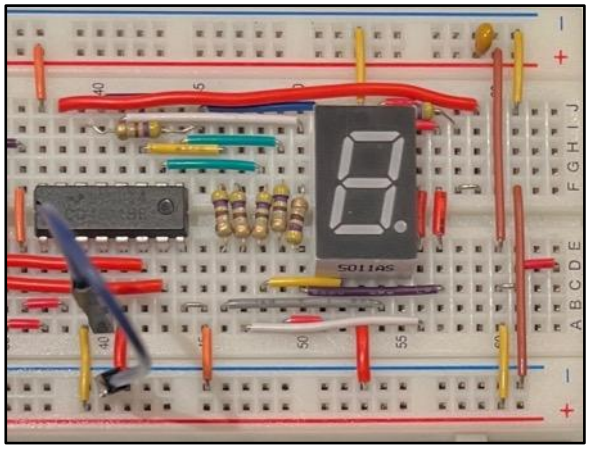
## Media




BCD Decoder Pins Properly Conditioned (BCD Count 7)



Display Test Low



Blank Input Low

## Reflection

While this section could easily have been combined with the seven-segment display section, I think that it made sense to separate it. Had they been the same section, I would have just wired up the display to the BCD Decoder, and accepted the fact that the BCD number was easily converted to a number on the display. Instead, when writing about the BCD Decoder, I wondered how it converted such a specific set of inputs to outputs. After doing reading through the datasheet and conducting some research, I stumbled upon a graphic on the ACES page showing the logic gate configuration inside the IC. This allowed me to actually think about the chip instead of painting by numbers. In addition to proving the writing process once again, this section taught me about logic gates and the composition of IC chips.

## F. Seven-Segment Display

### Purpose

The purpose of the Seven-Segment Display is to convert the decoded BCD signals to a human-readable output in the form of an LED-lit number display.

### Reference

Seven-segment display datasheet: http://darcy.rsgc.on.ca/ACES/Datasheets/5011-BS-200901A.pdf

Section description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#CCF

Section schematic: https://crcit.net/c/725bcdc27eab42849e905512cfe1bb2c

https://www.electronics-tutorials.ws/blog/7-segment-display-tutorial.html

### Procedure

The Seven-Segment Display is a rectangular block, with seven individual LED bars embedded into the top face of it, arranged in the shape of the digit "8" (see Figure 1). This allows it to display any single decimal digit; since each LED can be controlled independently of the others, by turning on or off certain LEDs, any number can be formed. For example, if all the LEDs are on, but segment G is low (see Figure 1), the number 0 is formed. The BCD Decoder controls the LEDs (see part E).

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | BCD Decoder (See Part E) |
| 1 | Seven-Segment Display |
| 7 | 470 Ω Fixed Resistor |
| 1 | 9V Power Source (Battery) |
| 1 | Breadboard |
| ~ | Wires |

There are two types of Seven-Segment Displays, both with slightly different methods of control: the first is called a *common anode* (CA) display. Common anode displays share their positive electrode, which means that an LED is active when its respective pin is grounded, or low. In this configuration, the current flows from the common anode pin (positive) to the output of the connected chip (or to ground via a resistor). This means that the chip is *sinking* the current. Sinking current is when current flows into an IC output.

The other type of Seven-Segment Display is called a *common cathode* (CC) display. Common cathode displays share their negative (ground) pin, making them the opposite of CA displays. In a CC display, an LED is active when its respective pin is connected to power, or high. In this configuration current flows from the IC (positive) to the common cathode pin (negative). This means that the chip is *sourcing* the current. Sourcing current is the opposite of sinking and occurs when current flows out of an IC output.

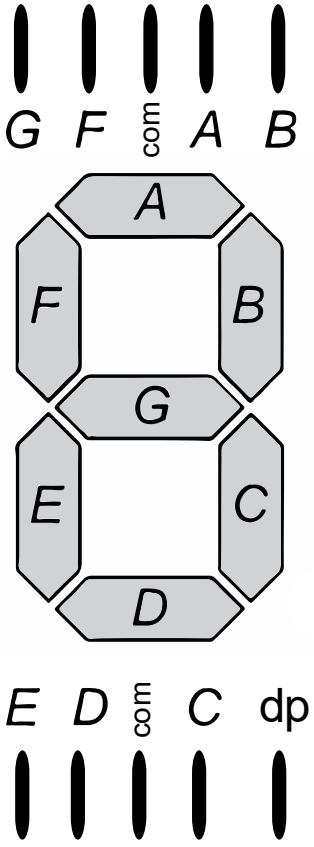



Figure 1. Seven-Segment Display

The Counting Circuit uses a common cathode display (see Figure 2). Each segment is connected to its respective output on the BCD decoder through a 470 Ω fixed resistor. Each pin having its own resistor is important; if there were one larger resistor on the common cathode pin, the display would not function properly.

When the count is at 8, all the display segments are lit up, meaning that the highest amount of current is drawn. The current can be calculated by adding each LEDs current together since they are connected in parallel, where $I = \frac{V}{R}$. Referring to the datasheet of the Seven-Segment Display, each LED consumes 2.1 V. To obtain the same current with one large resistor, the required resistance of the resistor would be $R = 470\ \Omega\ \times 7 = 3290\ \Omega$.

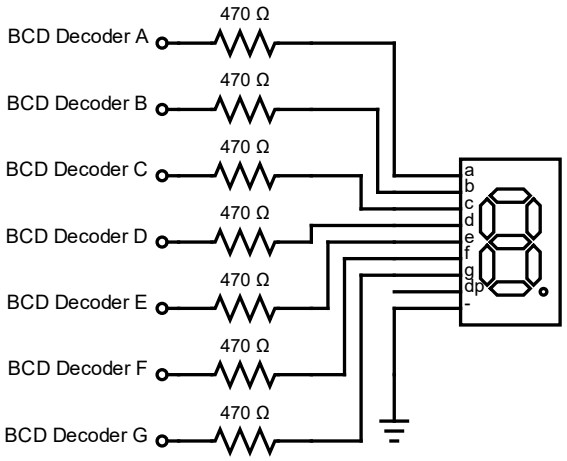



Figure 2. Seven-Segment Display with Input

Current in each segment can be expressed by the following equation: $I = \frac{2.1\ V}{3290\ \Omega}$ or 0.64 mA; since there are seven segments, however, the current would be 4.48 mA. When the number 1 is shown, only two segments are on. This means that the current is equal to $0.64\ mA \times 2 = 1.28\ mA$. Since current directly correlates with the brightness of an LED, in this configuration, every number has a slightly different brightness (see Figure 3).

**Figure 3.**
**Number of Segments Lit and Current**

| Decimal | Segments | Current |
|---|---|---|
| 0 | 6 | 3.84 mA |
| 1 | 2 | 1.28 mA |
| 2 | 5 | 3.2 mA |
| 3 | 5 | 3.2 mA |
| 4 | 4 | 2.56 mA |
| 5 | 5 | 3.2 mA |
| 6 | 5 | 3.2 mA |
| 7 | 3 | 1.92 mA |
| 8 | 7 | 4.48 mA |
| 9 | 5 | 3.2 mA |

This problem is solved by placing a resistor on each segment. The current of each segment can be expressed by the following equation: $I = \frac{2.1\ V}{470\ \Omega}$, or 4.46 mA. In this configuration, every segment has its own resistor, and the current remains the same in each segment regardless of which segments are lit.

The pin configuration of the BCD Decoder (4511) and the Seven Segment Display allows the components to be wired together very easily. All outputs on the 4511 are on the same side. Pins 6, 7, 9, and 10 on the display correspond to the first four outputs (see Figure 4). The final three outputs of the 4511 are on the opposite side of the display. If three spaces are left between the components, this avoids the need to cut any custom wires.

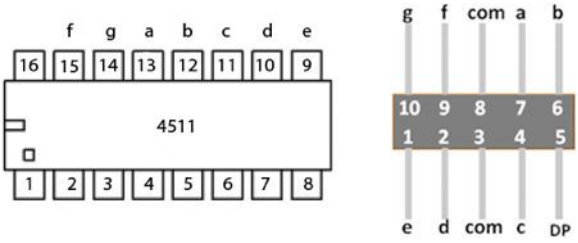
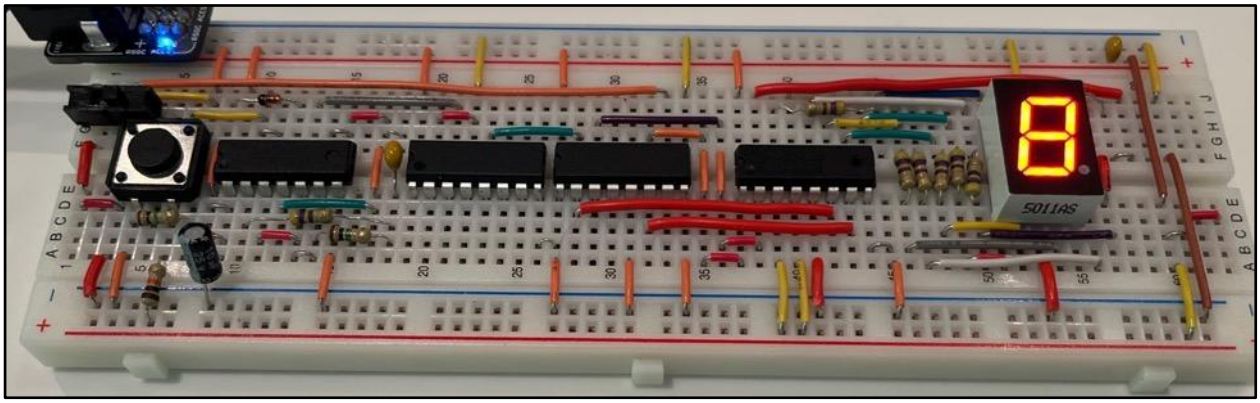
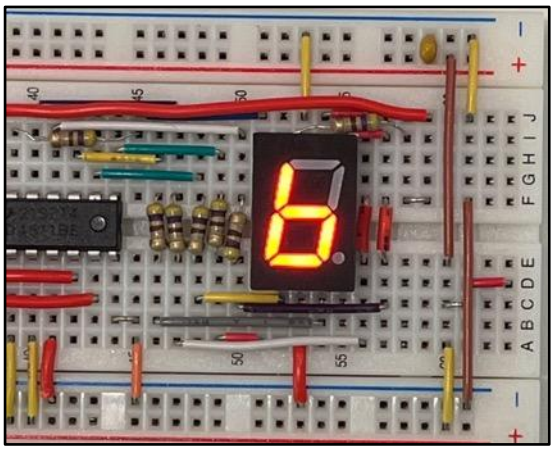



Figure 4. BCD Pinout to Display Input

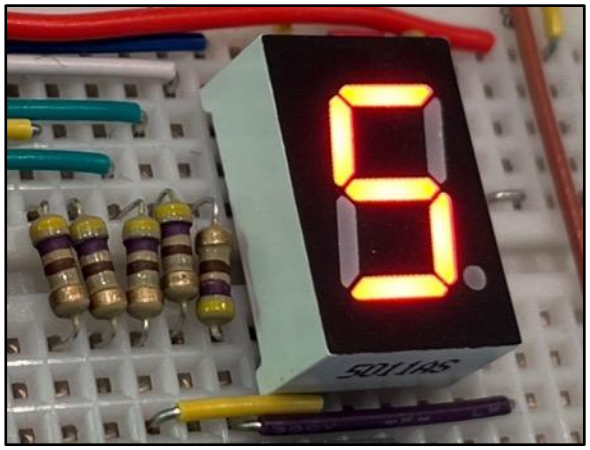
## Media



Final Breadboard Prototype (BCD Count 8)



Display Wiring Scheme



Resistors Spanning Gap and Display

## Reflection

When I first started building this section of the circuit, I thought it was going to be the easiest one. Easier than the Analog Input stage, even easier than wiring up the Decade Counter. Like many times before in this project, I have once again proven myself wrong. After the 15 or so minutes it took me to wire up the display, the circuit started counting. I pumped my fists in the air, excited that I had finally done it; when it got to seven, however, the display turned off, and the count reset to four after a few seconds. It continued doing this as long as I kept the circuit running. "Ok," I thought, "it can't be that big of a problem." Once again, I managed to prove myself wrong. I spent the next few days troubleshooting the circuit. I replaced my binary counter and rewired the entire circuit leading up to display to no avail. Eventually, I manually tested my BCD Decoder with jumper wires. I found that some numbers did not display properly. This brought me to the conclusion that my BCD Decoder was faulty. After much trial and error, I realized that I had swapped the f and g connections going to the display. Unlike my previous adventure with my attempt on a 2-Bit Magnitude Comparator (see Project 1.3), I had proper documentation, and I was able to fix it. Overall, I now understand how the entire breadboard circuit works, and counts from 0-9. I am now ready to obtain and solder my PCB.

## G. A Counting Circuit PCB

### Purpose

The purpose of A Counting Circuit PCB is to convert the temporary breadboard prototype (see part F) into a permanent form factor.

### Reference

Section description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#CCG

Section schematic: https://crcit.net/c/ff3b120e41e3455d985d6f4fc439021c

https://www.makerspaces.com/how-to-solder/

### Procedure

The Counting Circuit PCB is nearly functionally identical to the breadboard prototype (see part F); it does, however, contain many cosmetic refinements as well as small component changes: instead of a 10 µF capacitor, the PCB uses a 100 µF capacitor on the NGO. This means that when the button is pressed, the device runs longer. Instead of running for 7.5 seconds (see part B), the PCB runs for approximately 75 seconds after each button press.

Some cosmetic changes include a black disk capacitor to match the PCB color, built-in traces. As with any PCB, no wires are needed to connect components. The appropriate connections are internally made with traces, eliminating the need for wires

Another change made in the PCB version of A Counting Circuit is the use of isolated resistor networks (see Figure 1). Instead of having seven separate resistors for each segment of the display, one single in-line package (SIP) resistor network is used. SIP indicates that there is one line of pins. Each pair of pins provides a path through a 1 KΩ fixed resistor (see Figure 1).

For power delivery, a 2-position terminal block is used (see Figure 2). In this terminal block, there are two leads and two holes. Each hole is connected to one of the leads. This allows the power delivery device to be swapped in and out. The use of a terminal block elevates the professionality of the PCB to a further level.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | Custom PCB |
| 1 | 0.1 µF Disk Capacitor |
| 1 | 100 µF Electrolytic Capacitor |
| 1 | 14-pin IC Socket |
| 3 | 16-pin IC Socket |
| 2 | 1 KΩ (Isolated) Resistor Network |
| 1 | 2-pos 2.54 mm Terminal Block |
| 1 | PBNO |
| 2 | SPDT Slide Switch |
| 2 | 1 KΩ Fixed Resistor |
| 2 | 1 MΩ Fixed Resistor |
| 1 | 470 Ω Fixed Resistor |
| 1 | Signal Diode |
| 1 | Seven-Segment Display |
| ~ | Solder |

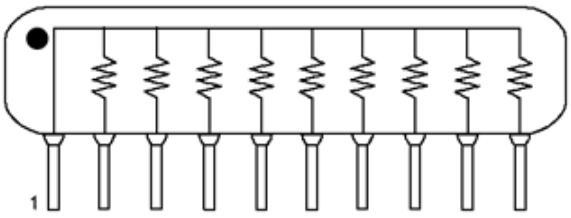
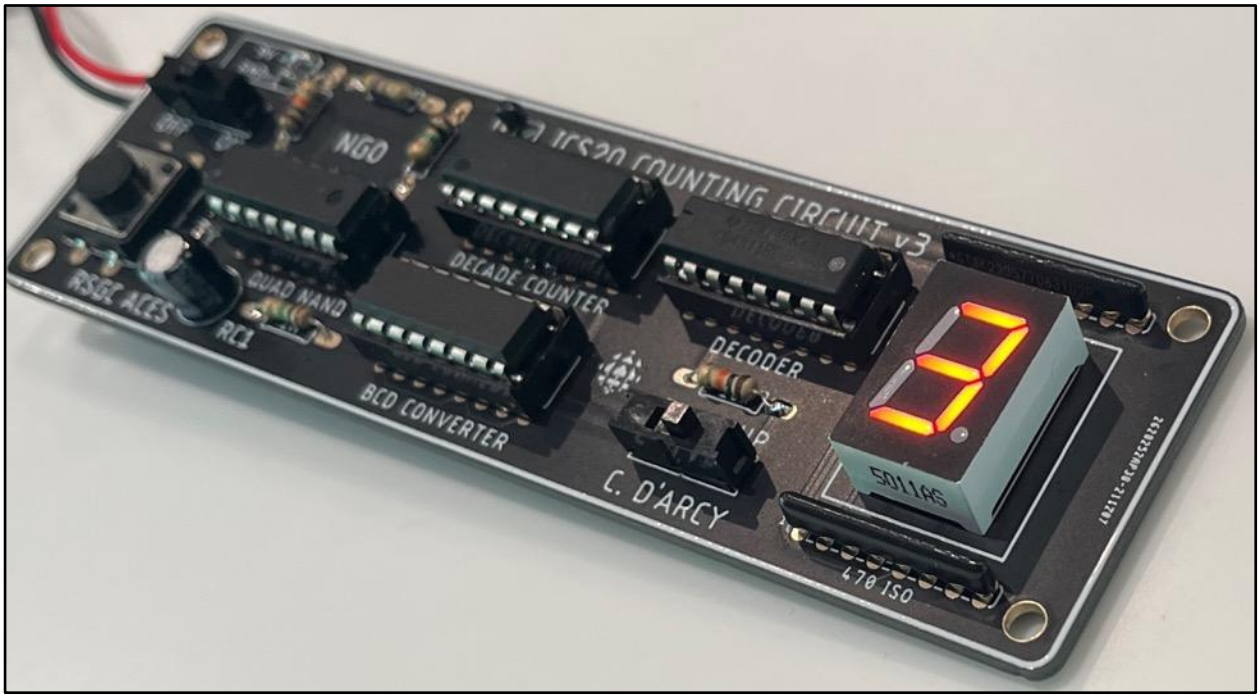


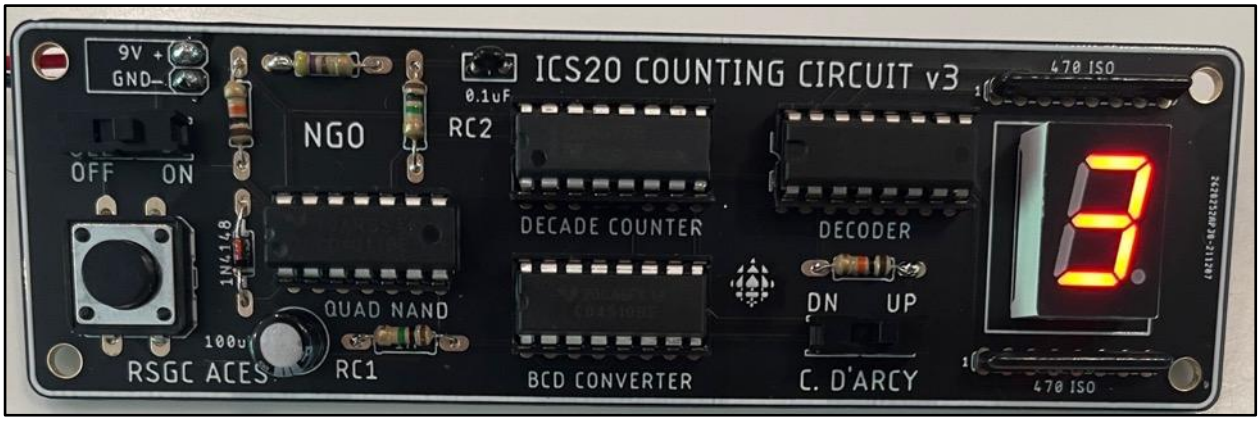

Figure 1. Isolated Resistor Network

Figure 2. 2-Position Terminal Block

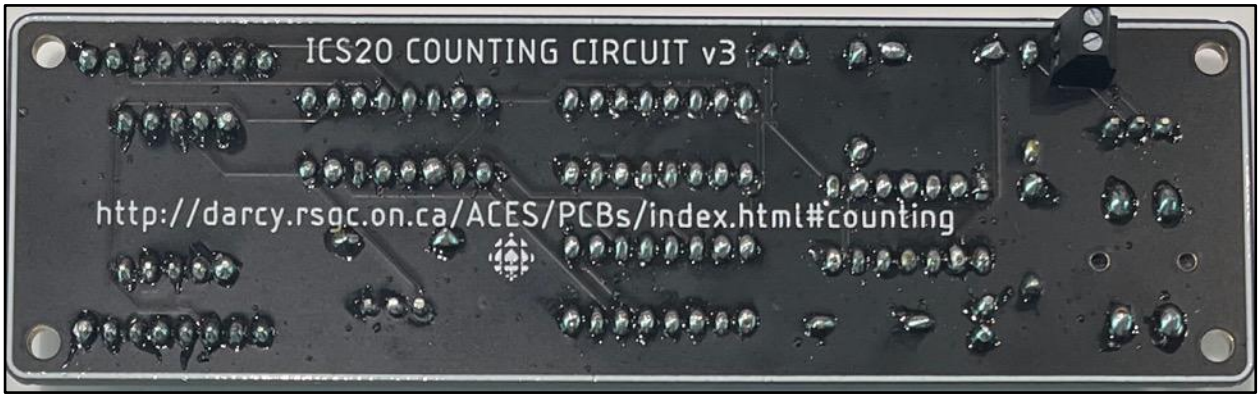
## Media



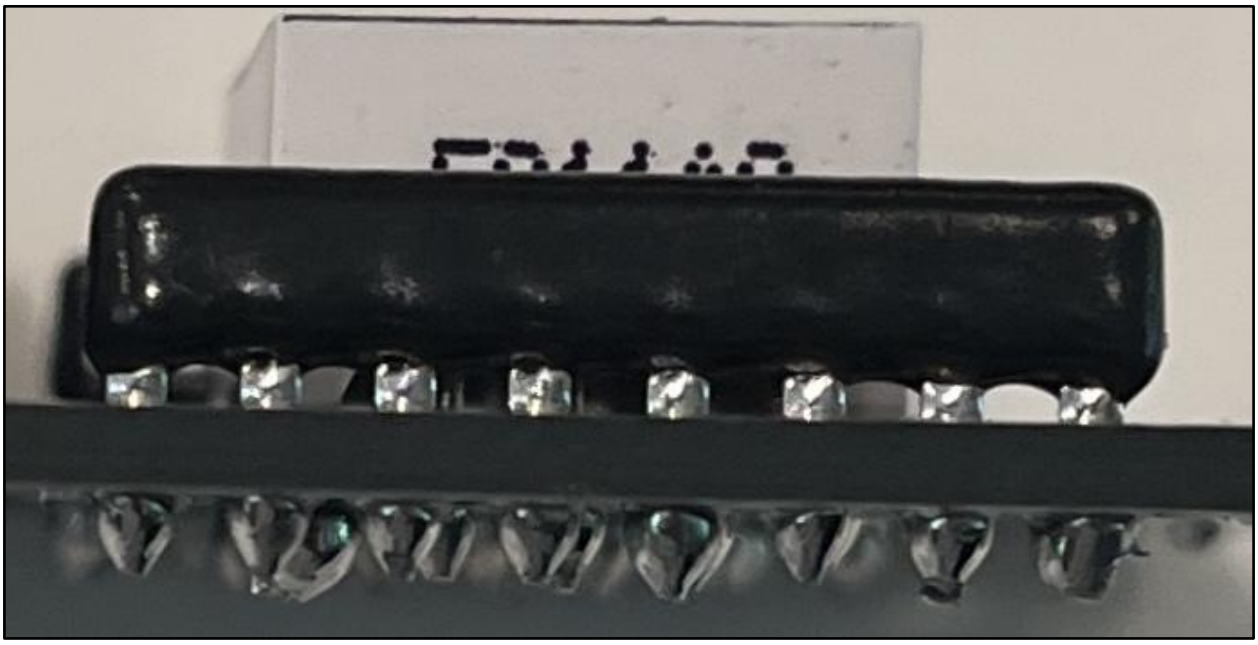

Counting Circuit PCB (BCD Count 3)




Counting Circuit PCB Full View (Front)




Counting Circuit PCB Full View (Back)



Isolated Resistor Network (SIP)

BCD Count 0

BCD Count 9

## Reflection

Of all the sections of this project, it was the PCB that went the most smoothly. I underestimated the difficulty of nearly every other section; this was the first section who's difficulty I correctly estimated. Everything worked the first time on this stage, aside from my NAND chip. For some reason, my NAND chip went bad after I had already soldered it (without the chip in). I think that it's possible that the chip may have bounced around in my kit and malfunctioned somehow. It's not the worst issue that could have befallen me; only the duration gate went bad. This means that no matter the charge of the capacitor, the circuit constantly runs. Overall, this section went relatively smoothly for me, and I am excited to add the case to the device and fully complete the aesthetic.

## H. A Counting Circuit PCB Case

### Purpose

The purpose of A Counting Circuit PCB Case is to house the Counting Circuit PCB (see part G), completing the finished, permanent aesthetic of the device.

### Reference

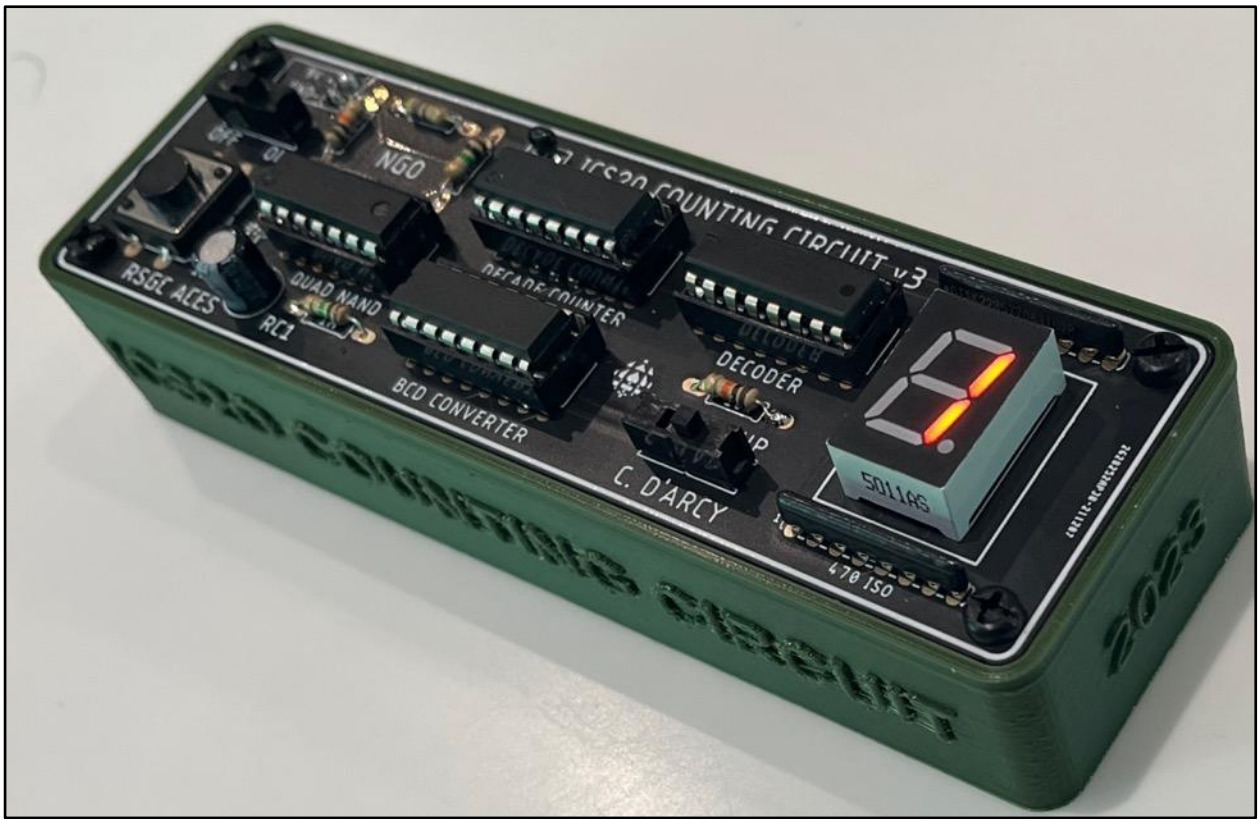
Section description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#CCH

### Procedure

The Counting Circuit PCB Case is a 3D printed plastic case, designed to house the Counting Circuit PCB, as well as a 9 V battery. The battery snap and battery fit perfectly into the center of the case, and the wires run into the terminal block of the PCB. Electrical tape is placed over top of the battery to insulate it from the PCB, preventing shorts. Finally, the PCB is screwed into the case, securing it into place.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | Counting PCB (see part G) |
| 1 | Counting Circuit Printed Case |
| 1 | 9 V Battery |
| 1 | 9 V Battery Snap |
| 4 | Black Screws |
| ~ | Electric Tape |

### Media




Counting Circuit PCB Case (BCD Count 1)


Counting Circuit PCB Case Side View

Battery Holder (Empty)


Battery Holder (Populated)

## Reflection

This final section of the report was, once again, a section that I didn't think should have gotten its own section initially. When I thought this, I knew in the back of my mind that I would end up realizing why it had been separated from the PCB itself. I now do indeed understand it; the case adds a final refinement to the device. It binds and hides the battery, making it a truly complete device.

# I. Dual-Digit Counting Circuit

## Purpose

The purpose of the Dual Digit Counting Circuit is to extend the functionality of the regular counting circuit, and count up to 99.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEL3M/Tasks.html#counting

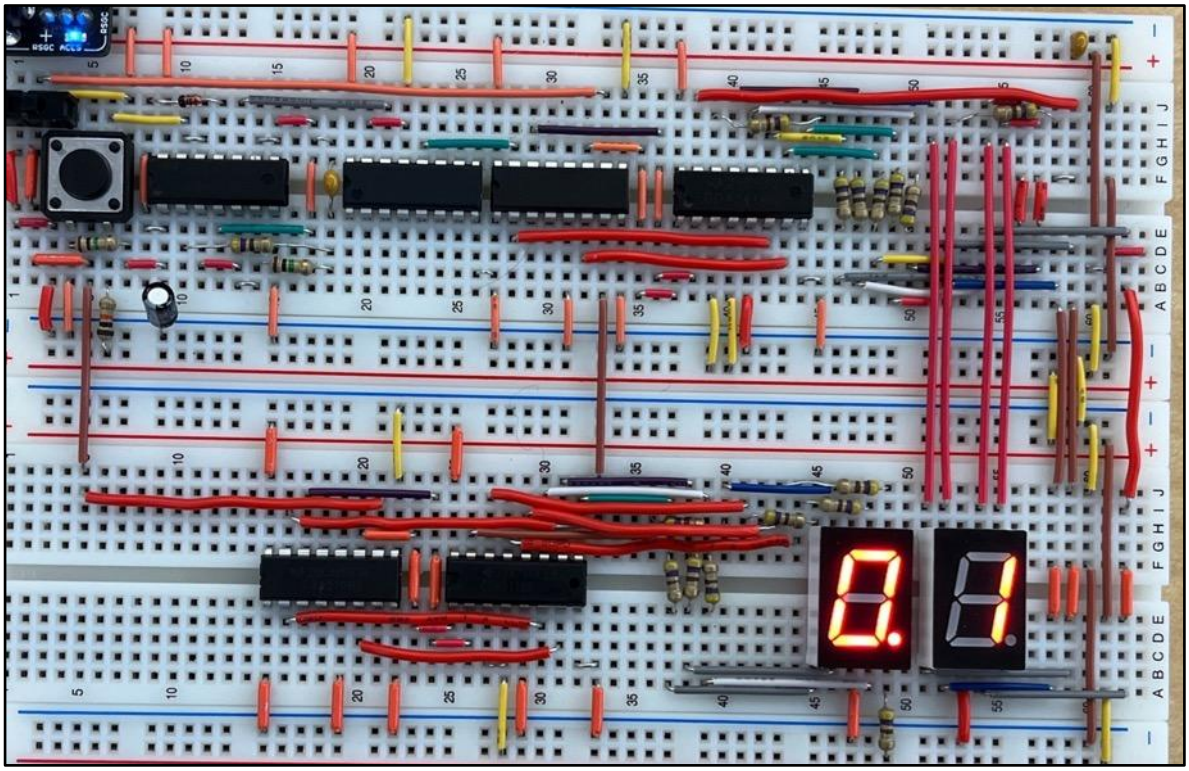
https://www.build-electronic-circuits.com/4000-series-integrated-circuits/ic-4511/

## Procedure

The Dual-Digit Counting Circuit works in a nearly identical fashion as the Single-Digit Counting Circuit. For dual counting functionality, a binary counter, decoder and display are built. Instead of running off the clock signal of the Decade Counter (see part C), the second digit's clock is driven by the carryout pin on the first BCD Counter. This means that every time the first counter reaches 0 (or 9 when counting down), one clock cycle on the carryout pin takes place. This allows the second digit to count at a rate 10 times slower than the first.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | Counting Circuit (see part F) |
| 8 | 470 Ω Fixed Resistor |
| 1 | Seven-Segment Display |
| 1 | 4510 BCD Counter |
| 1 | 4511 BCD Decoder |
| 1 | 0.01 µF Disk Capacitor |
| ~ | Wires |

## Media



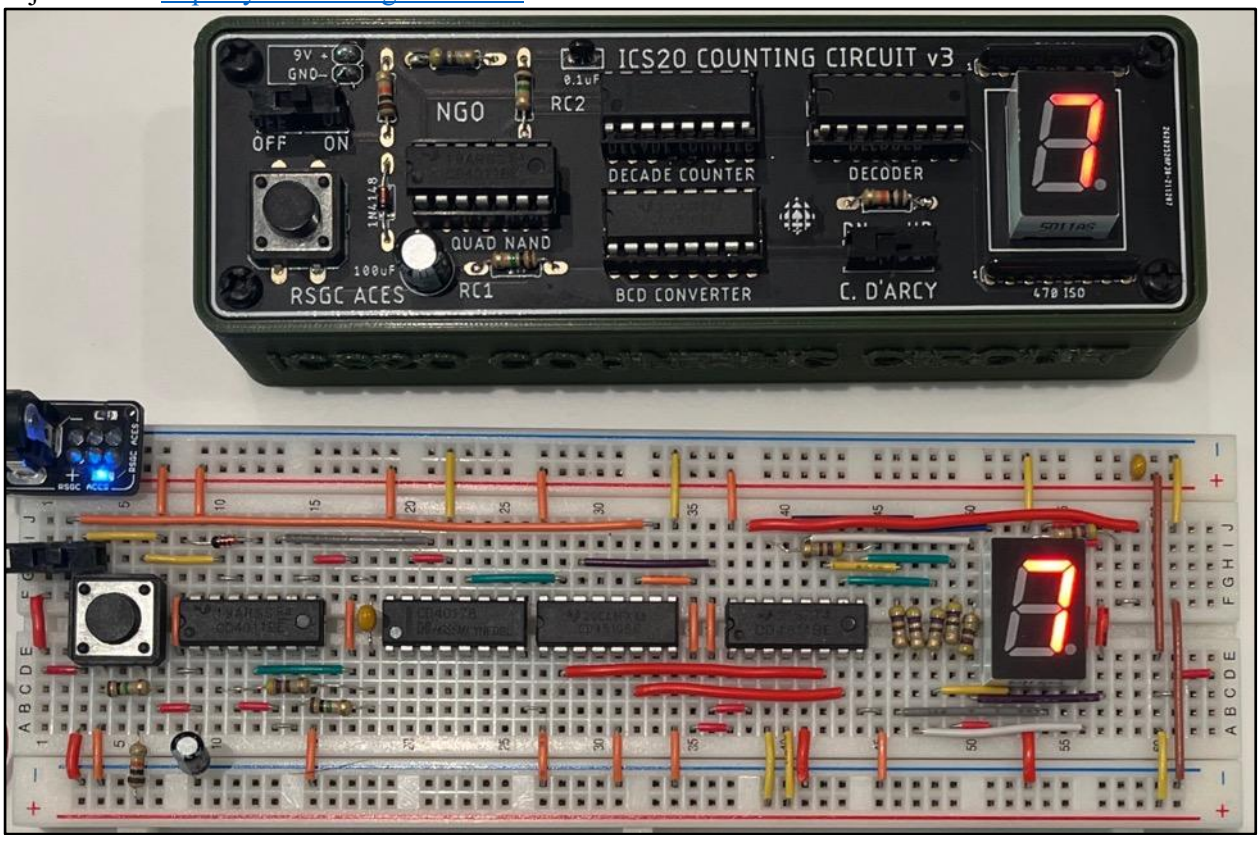
Dual Digit Counting Circuit

## Reflection

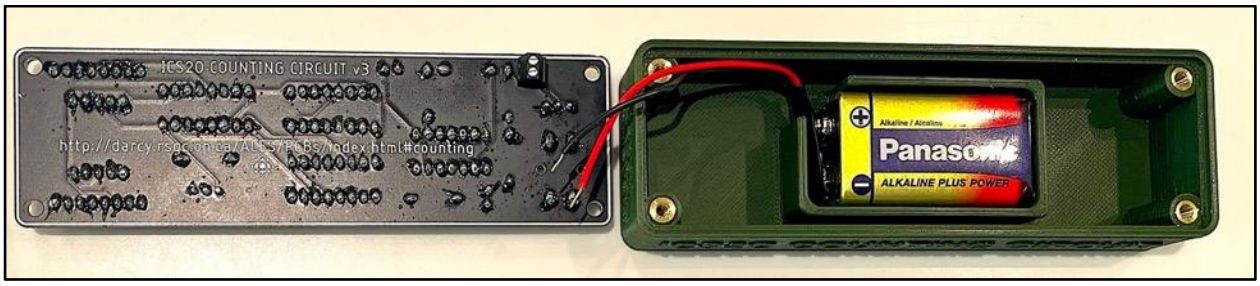
This section was a last-minute addition to the report, but it was well-worth the trouble. Until its addition, I did not understand the carryout function of the Binary Counter. Now, I can say that I do.

## Media

Project video: https://youtu.be/tGgYJRA9Ft4




Finished PCB Device Beside Single-Digit Breadboard Prototype




Bottom of PCB, Battery, and Case

## Reflection

This has been an extremely interesting project for me. When I first found out that I had a week to essentially do 8 DERs, I never thought I was going to finish. This is most likely the most time I have ever put into a school project, at least for a short-term one, but the result is astonishing. This project has taught me the value of working hard on something. After spending many hours on this project, I am happy to say that my experience in this course has been successful. I still remember the first class, when I was scared to answer questions, and take the risk of being wrong. I have come a long way since then, taking the risk whenever presented the opportunity. Overall, this has been a great year, and I am happy to say that I think this course came around at the right time of my life. I look forward to more technical (mis)adventures next year.

# ICS3U-E

# Project 2.1 Digital-to-Analog Conversion

## Purpose

By converting digital signals to analog voltages, the Digital-to-Analog Converter (DAC), in the form of an R-2R ladder allows digital devices to interface with real-world analog components and signals.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI3M/Tasks.html#R2R

Project schematic: https://crcit.net/c/8c6df86c73e241d08b7e14129bb83f63

## Theory

The R-2R DAC is a circuit with a certain number of bits, each adding its own binary-weighted voltage to the circuit's output. The voltage can be calculated using the *superposition theorem*. The superposition theorem states that "the voltage across (or current through) an element in a *linear circuit* is the algebraic sum of the voltage across (or currents through) that element due to each independent source acting alone." A linear circuit is a circuit that contains no *non-linear components*. A non-linear component is a component whose parameters are not constant with respect to voltage and current. This effectively means that in order to find the voltage at any point in a circuit using the superposition theorem, the effect of each power source must be considered, individually at first, then added together for the net voltage. This is done by turning off all power sources except one, for each individual power source, calculating the voltage at the desired point using Kirchhoff's Voltage Law and Ohm's law, then adding each result together. A voltage source is turned off by replacing the source with a short circuit, whereas a current source is turned off by replacing the source with an open circuit.

## Procedure

The R-2R DAC (see Figure 1) makes use of the voltage dividing properties of resistors (see Project 1.1). In the R-2R configuration, an array of paired resistors, where the resistance of the first is exactly double (or half) of the second, are configured in combination with switches to allow the circuit to output a voltage between 0 and the source voltage. In this circuit, the number of different possibilities pertains to the number of bits (switches) the circuit includes; a greater number of bits is analogous to a higher resolution. Each additional switch adds two resistors as well: one with a value of R Ω, and one with a value of 2R Ω. The R-2R DAC can be scaled to any desired number of bits.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | Rocker DIP Switch Bank |
| 9 | 10.2 KΩ Fixed Resistor |
| 7 | 5.1 KΩ Fixed Resistor |
| 1 | 470 Ω Fixed Resistor |
| 1 | Bussed Resistor Network |
| 1 | LM741 Operational Amplifier |
| 1 | 5mm LED |
| 1 | 9V Battery |
| ~ | Wires |

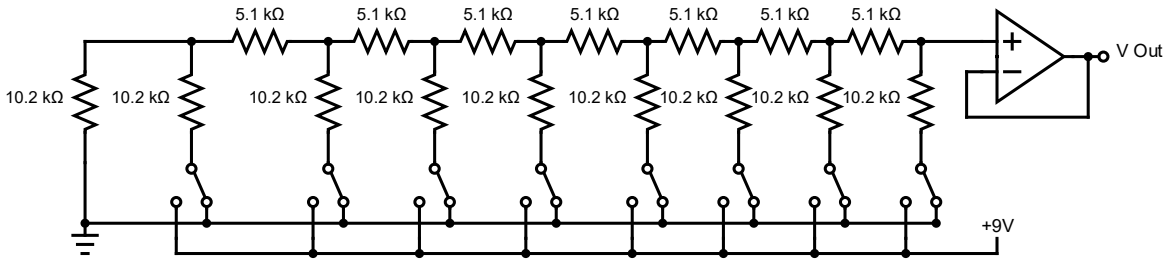



Figure 1. R-2R DAC

Thevenin's Theorem states that any linear circuit (see Theory section) can be simplified to an equivalent circuit consisting of a voltage source and a single series resistor. It allows the circuit to be "cut" at any point, and have everything to the left of the cut replaced with a voltage source equal to the open-circuit voltage at the cut point, and a series resistor with a resistance equal to the open circuit resistance with all voltage sources shorted. Thevenin's Theorem can be used to simplify the R-2R DAC to a resistor with a value of 5.1 KΩ, or R as well as an op amp (see Figure 2).

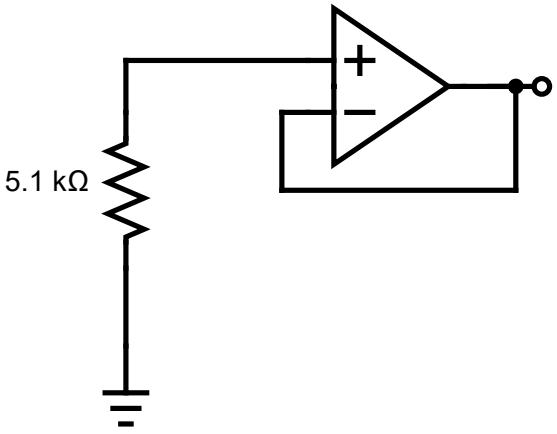



Figure 2. Thevenin Equivalent Circuit

Assuming each bit is grounded, the first two resistors (from left to right, see Figure 1) are in parallel, meaning they can be simplified to a singular 5.1 KΩ resistor to ground (see Figure 3).

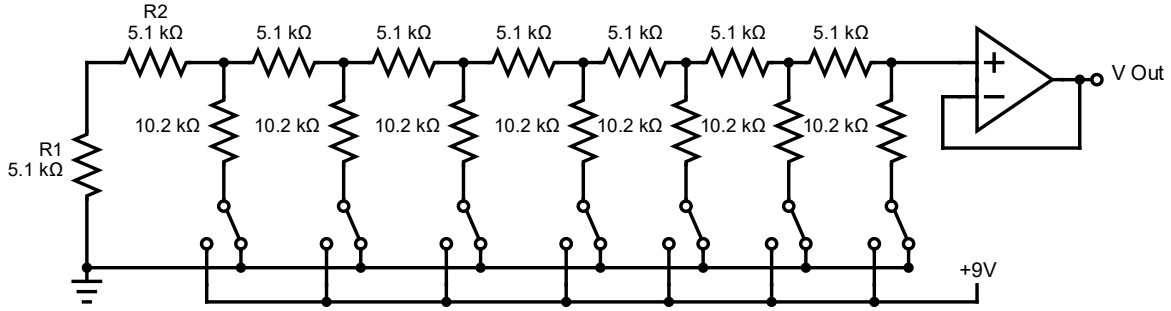



Figure 3. Simplified R-2R DAC

In Figure 3, R1, which came from two resistors pictured in Figure 1, is in series with R2, meaning it can be replaced with a 10.2 KΩ resistor. This is effectively the same as the original circuit, but with one less bit. This is what allows the R-R2 DAC to be scaled to any number of bits; no matter how many bits are added, the circuit can always be simplified to a resistor with a value of R Ω.

To calculate the output voltage of the ladder, Thevenin's Theorem can be combined with the superposition theorem (see Theory section). This means that at any time, to calculate the output voltage, only one switch is active. The total output voltage is equal to the sum of the voltage contributed by each bit. When the leftmost switch (see Figure 1) is high, the leftmost bit (including its resistors) can be replaced with its Thevenin's Equivalent (see Figure 4).

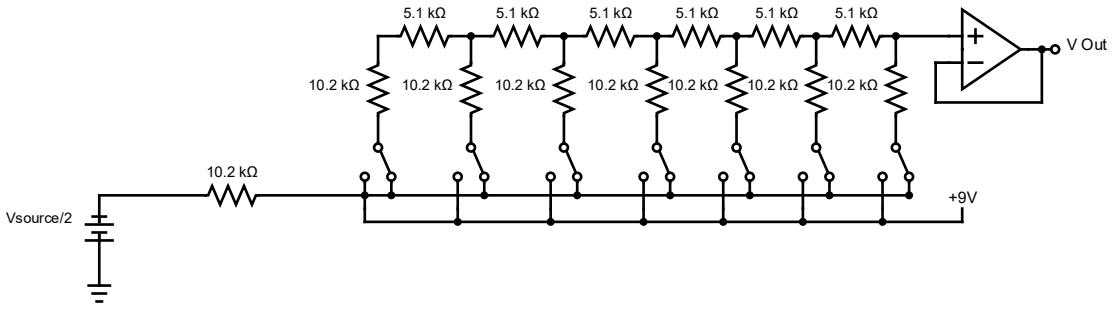



Figure 4. Leftmost Bit High Thevenin's Equivalent Circuit

In this configuration, the voltage is divided by two, due to the fact that it is being measured between two equal resistors; therefore, it acts as a voltage divider. $V_{out} = \frac{9\ V \times 5.1\ K\Omega}{(5.1\ K\Omega + 5.1\ K\Omega)} = 4.5\ V$ This process can be repeated by "cutting" the circuit and replacing with the Thevenin's Equivalent until it reaches the output of the circuit. Each time the circuit is cut, the voltage is reduced by half. In an 8-bit ladder, the leftmost bit (see Figure 1), or least significant bit (LSB), contributes $\frac{1}{2^8}\ V_{source} = \frac{1}{256}\ V_{source}$ to the output due to being divided in half across 8 different bits. When this process is repeated with bits further to the right side of the schematic, it is divided in half less times; therefore, the bit furthest to the right is the most significant bit (MSB) and contributes $\frac{1}{2} V_{source}$ to the output of the circuit (see Figure 5). Since $\frac{1}{256} + \frac{1}{128} + \frac{1}{64} + \frac{1}{32} + \frac{1}{16} + \frac{1}{8} + \frac{1}{4} + \frac{1}{2} = \frac{255}{256}$ the voltage value of the LSB is lost in the DAC.

**Figure 5: Voltage Contribution Per Bit**

| Bit (LSB-MSB) | Contribution (V) |
|---|---|
| 1 | $\frac{1}{256} V_{source}$ |
| 2 | $\frac{1}{128} V_{source}$ |
| 3 | $\frac{1}{64} V_{source}$ |
| 4 | $\frac{1}{32} V_{source}$ |
| 5 | $\frac{1}{16} V_{source}$ |
| 6 | $\frac{1}{8} V_{source}$ |
| 7 | $\frac{1}{4} V_{source}$ |
| 8 | $\frac{1}{2} V_{source}$ |
| **Total** | $\frac{255}{256} V_{source}$ |

## Rocker DIP Switch Bank

For a more compact design, the R-2R ladder can be built using a singular DIP Rocker DIP Switch Bank (see Figure 1). These switches act as single pole, single throw switches, unlike the original circuit, which uses single pole, double throw switches. In this configuration, a pull-down resistor is required in order to keep the logic-low voltage at 0 V instead of floating. A bussed resistor network (see Figure 2) can be used to pull each switch down. A bussed resistor network consists of an array of resistors, all with one common pole. In this case, the common pole goes to ground, pulling each switch down.


Figure 1. Rocker DIP Switch Bank

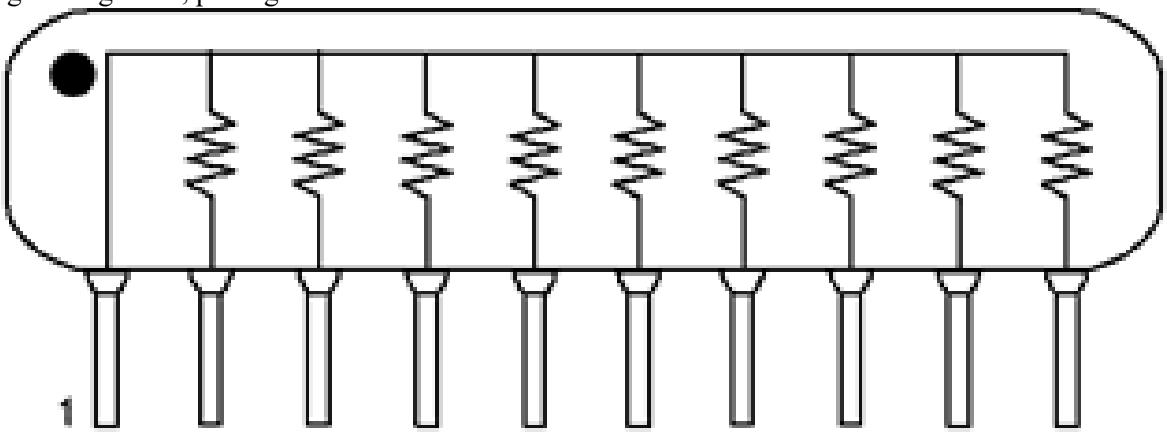



Figure 2. Bussed Resistor Network

## Operational Amplifier

An *operational amplifier* (op amp) can be added to the output of the R-2R ladder to make the voltage changes more apparent, and to remove the circuit's impedance of 5.1 KΩ from the output. An op amp is a device that can amplify weaker electric signals. It consists of two inputs and an output. One input is inverting (-), and one is non-inverting (+).

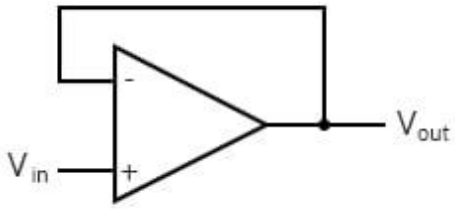
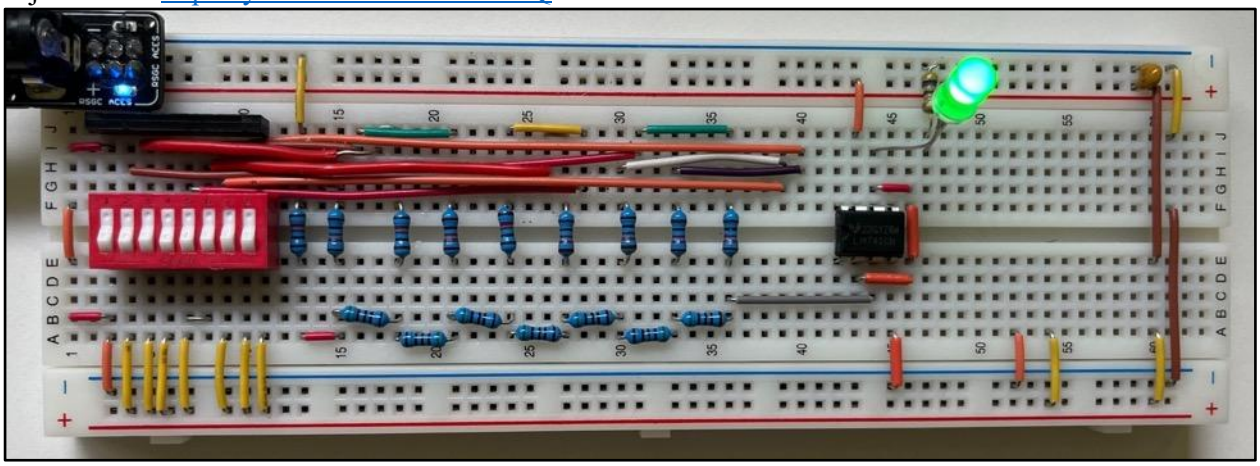



Figure 3. Op Amp with Feedback

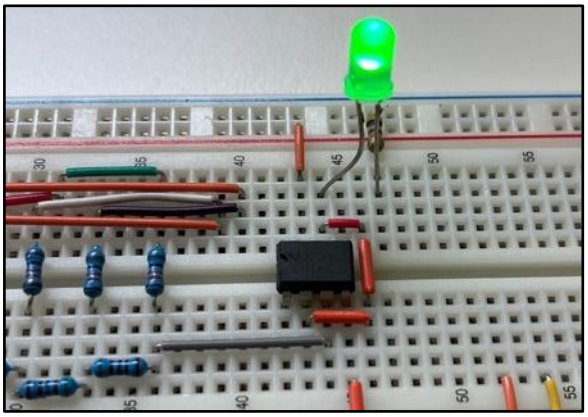
When the inverting input is connected to the output (see Figure 3), a feedback loop is created; an op amp tries to keep the two inputs at the same voltage. This effectively means that an op amp with a feedback loop between the output and inverting input will have an output voltage matching the voltage presented on the non-inverting input, without any impedance present. It is able to do this since it has its own separate power supply, consisting of a negative and positive supply of equal magnitude.

## Media

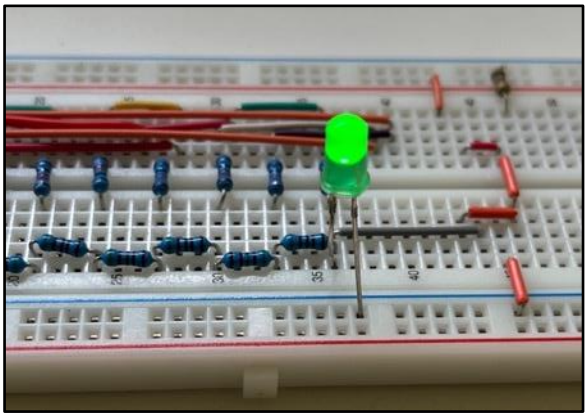
Project video: https://youtu.be/v86UMGtVO5Q



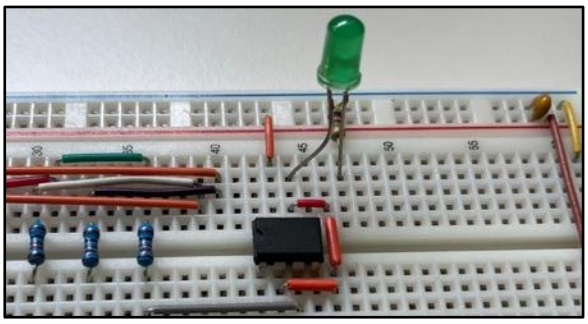
R-2R DAC



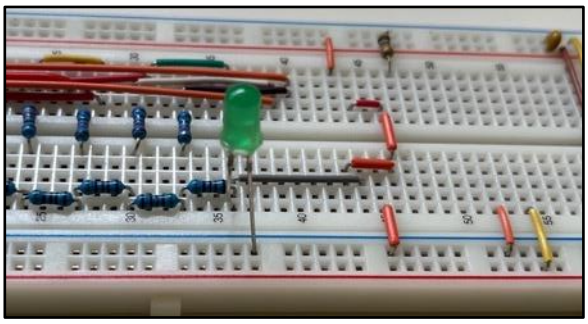
All Bits High (With Op Amp)



All Bits High (Without Op Amp)



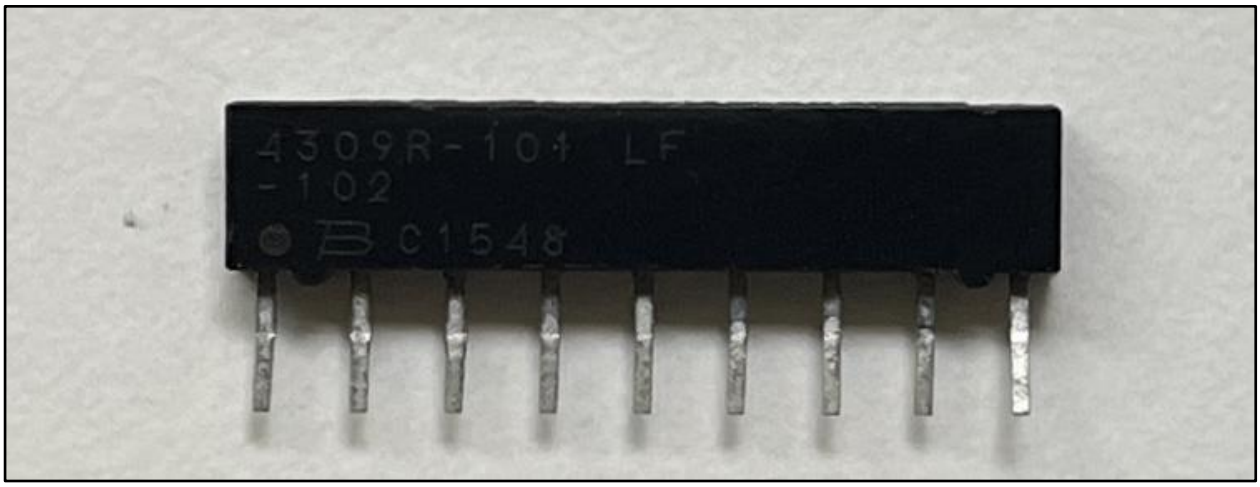
All Bits Low (With Op Amp)



All Bits Low (Without Op Amp)


Source Voltage


Output Voltage (All Bits High)




Bussed Resistor Network

## Reflection

I know I say this every time, but this has actually been a pretty interesting project for me. I missed quite a few classes since I was at camp and cross country, so I got a pretty late start on the project. When I started, I had pretty little time, and no idea where to start. I first started to try and learn about the superposition and Thevenin's equivalent theorems. It took me a while to finally get a grip on these things and understand why the DAC functioned the way it did, but it taught me some important lessons. Thevenin's equivalent theorem essentially allowed me to break down the problem at hand into smaller problems and solve them using other concepts I already knew. As cheesy as this sounds, I think this is not just an important concept in the context of computer engineering, but in life as well. Most of the time, highly complex problems can be solved easily with knowledge we already have, we just need to change our perspective (yes, I know it's cheesy, but still). Of course, due to my late start, time management was not exactly on point for this project. For the next project, I'm hoping that I will be able to make a perf board equivalent of the circuit, or at least some sort of permanent/soldered version. Regardless, I think that this project has been a successful first of the year, and I hope that it will set the tone for the rest of the year.

# Project 2.2 The 555 Time Machine

## Purpose

In addition to debouncing pushbutton input signals, the purpose of the 555 Time Machine is to output a continuous and autonomous square wave.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI3M/Tasks.html#555

Project schematic: https://crcit.net/c/7b323485091a41eeb7126623d6817a26

555 datasheet: https://www.ti.com/lit/ds/symlink/lm555.pdf

https://youtu.be/kRlSFm519Bo

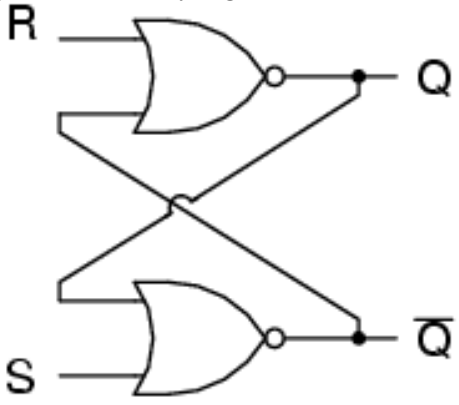
https://youtu.be/KM0DdEaY5sY

## Theory

The 555 Time Machine is an extremely useful IC chip that has many applications, mainly as a clock signal generator. It can be configured in both *astable* and *monostable* modes. Astable mode refers to a process where there is no stable state; the output continuously and autonomously switches between high and low states. It generates a square wave. Monostable mode refers to a process where there is one stable state. It remains in the low state until an input is received, at which point the output goes high, for a set amount of time.

The 555 Time Machine also employs a set-reset (SR) Latch (see Figure 1). An SR Latch is a sequential logic circuit that is capable of storing, or latching onto, an input as high or low. It does this by introducing feedback to NOR gates; each NOR gate has one input connected to the output of the other NOR gate, and the other input connected to a pushbutton. The effect of this is that the two outputs are the inverse of each other (other than when both inputs are high, in which case the output is invalid). The state of the outputs depends on which input was most recently high. This allows the SR Latch to reliably store a value, as well as reset the stored value.

In addition to an SR Latch, the 555 Time Machine circuit uses 2 op amps as comparators. In this configuration, the op amp compares the voltage present on the inverting (-) input to the voltage on the non-inverting (+) input. If the voltage on the non-inverting input is higher than the voltage presented on the inverting input, the comparator will output a high signal. If it is lower, it will output a low signal. Unlike when it is used as an amplifier, in this configuration, there is no feedback loop created, therefore the gain is extremely high.




| S | R | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0 | latch | latch |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

Figure 1. SR Latch and its Truth Table

## Procedure

The 555 Time Machine is a multi-purpose IC chip, making use of voltage dividers, op amps as comparators, and an SR Latch. In astable mode (see Theory section), the 555 timer generates a square wave.

When in astable mode, to generate a square wave, the 555 timer utilizes the voltage-dividing properties of resistors to create two reference voltages in addition to the supply voltage and ground. With three resistors in series between 5 volts and ground (see Figures 1 and 2), the two intermediary points can be calculated with the following formula: $V_{out} = \frac{V_S \times R_2}{R_1 + R_2}$. This means that V1 is equal to $\frac{5V \times 10.2\ K\Omega}{5.1\ K\Omega + 10.2\ K\Omega} \approx 3.33\ V$ and V2 is equal to $\frac{5V \times 5.1\ K\Omega}{10.2\ K\Omega + 5.1\ K\Omega} \approx 1.67\ V$.

When the circuit is first switched on, the reset comparator (see Figure 2) which is comparing the reference voltage of 1.6 V on the non-inverting input, to a voltage of 0 V on the inverting input, is high. Since 1.6 V is greater than 0 V, the set comparator will output a high signal, resetting the SR Latch. This low signal is stored until the set pin is high. Since the LED is connected to the inverted output, it is on in this state.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | 1 MΩ Potentiometer |
| 1 | 5 mm LED |
| 1 | 470 Ω Fixed Resistor |
| 4 | 5.1 KΩ Resistor |
| 1 | 10.2 KΩ Resistor |
| 1 | 2N3904 NPN Transistor |
| 1 | 1 µF Electrolytic Capacitor |
| 1 | LM358N Dual Op Amp IC |
| 1 | CMOS 4001 NOR IC |
| 1 | 5V Power Source |
| ~ | Wires |

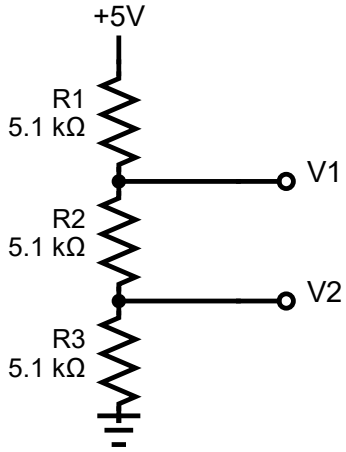



Figure 1. Voltage Dividers in the 555 Timer

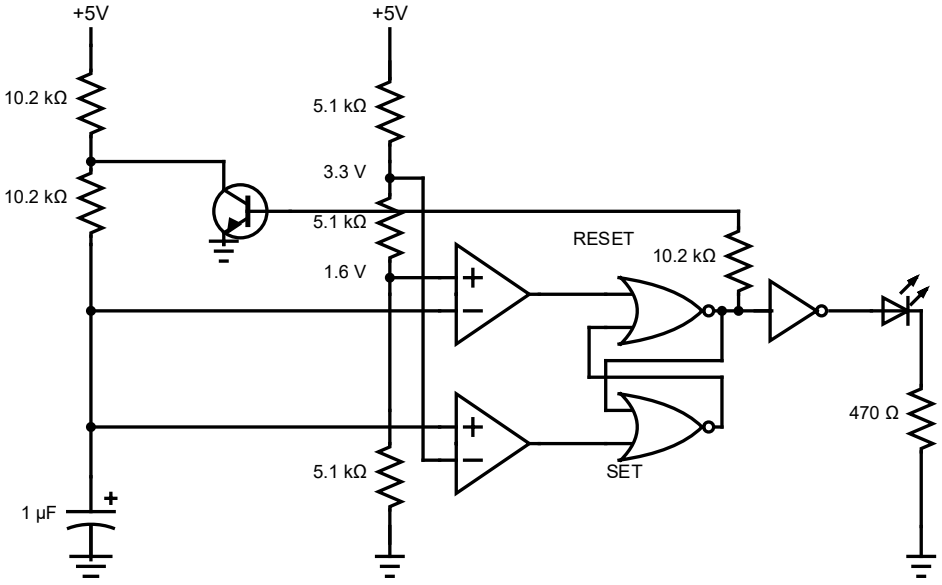



Figure 2. The 555 Time Machine in Astable Mode

When the circuit is switched on, the 1 μF capacitor (see Figure 2) begins to charge up. Once it charges beyond the reference voltage of 1.6 V, the reset comparator outputs a low signal, since the voltage presented on the inverting input is no longer less than the voltage presented on the non-inverting input. At this point, the output remains the same; the SR Latch has not been set yet, and its output will not change until the set pin is high.

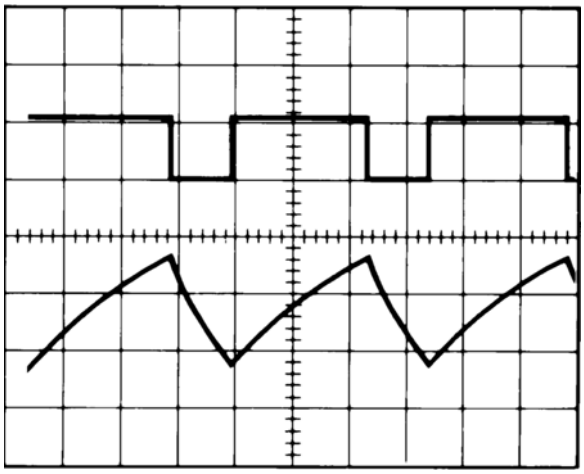


Figure 3. Capacitor Charge vs Output

Once the capacitor charges beyond the second reference voltage of 3.3 V, the set comparator outputs a high signal, since the voltage of the capacitor presented on the non-inverting input is greater than the reference voltage of 3.3 V presented on the inverting input. At this point, the output of the SR Latch has changed from low to high, causing the LED, connected to a NOT gate, to switch off. When this happens, the NPN discharge transistor, with its base connected to the SR Latch output, switches on, allowing current to flow through it.

This causes the capacitor to discharge slowly, as the transistor provides a path between the positive electrode of the capacitor and ground through a 10.2 KΩ resistor. When the capacitor discharges below 3.3 V again, the output remains the same, since the SR Latch has latched on; when the capacitor discharges below 1.6 V, however, the SR Latch is reset, switching the LED back on, and the transistor, off. This is what causes the cycle to repeat continuously, and autonomously. Figure 3 shows how the SR Latch helps convert the curved capacitor charge-discharge cycle to a square wave.

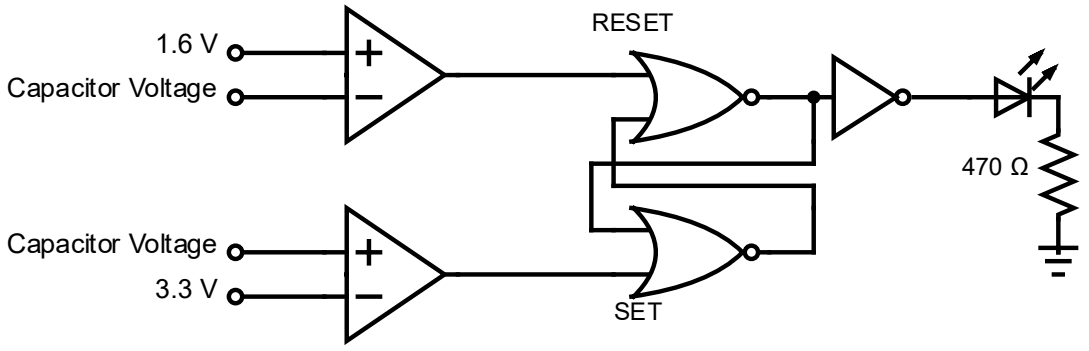



Figure 4. Voltages Compared by Op Amps in the 555 Time Machine

For the SR Latch to be in an invalid state, both the set and reset pins must be high (see Theory section). In the 555 timer's configuration, the SR Latch cannot be put into an invalid state; for the reset comparator to be high, the capacitor voltage must be less than 1.6 V, and for the set comparator to be high, the capacitor voltage must be greater than 3.3 V (see Figure 3). Since the capacitor cannot be charged greater than 3.3 V and less than 1.6 V simultaneously, the set and reset pins cannot both be high at the same time.

The cycle time is calculated using values R1, R2, and C (see Figure 5). The time to charge the capacitor (when the output is high) is calculated by $t_1 = 0.693(R1 + R2)C$. This formula takes the time constant and multiplies it by 0.693. If the capacitor were charging to full, the constant would be 5, which is how many time constants it takes for a capacitor to reach (almost) its full charge; the capacitor, however, only needs to charge to around 3.3 V. The constant 0.693 is provided by the 555 timer's datasheet. In this case, the charge time is equal to $0.693(10.2\ K\Omega + 10.2\ K\Omega)(1\ \mu F)$, which is equal to approximately 14 milliseconds.

The discharge time is calculated by $t_2 = 0.693(R2)C$. The formula for discharge time is the same as that of the charge time, with one difference. The capacitor only discharges through R2, not R1 and R2; therefore, the formula includes only R2. The time to discharge is equal to $0.693(10.2\ \text{K}\Omega)(1\ \mu F)$, which is equal to approximately 7 milliseconds. This is exactly half of the charge time, since when discharging, the capacitor experiences exactly half the resistance.

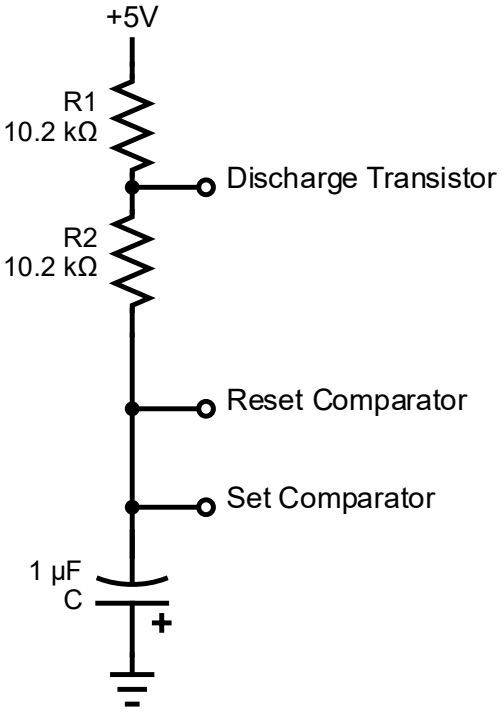



Figure 5. Timing Resistors and Capacitor

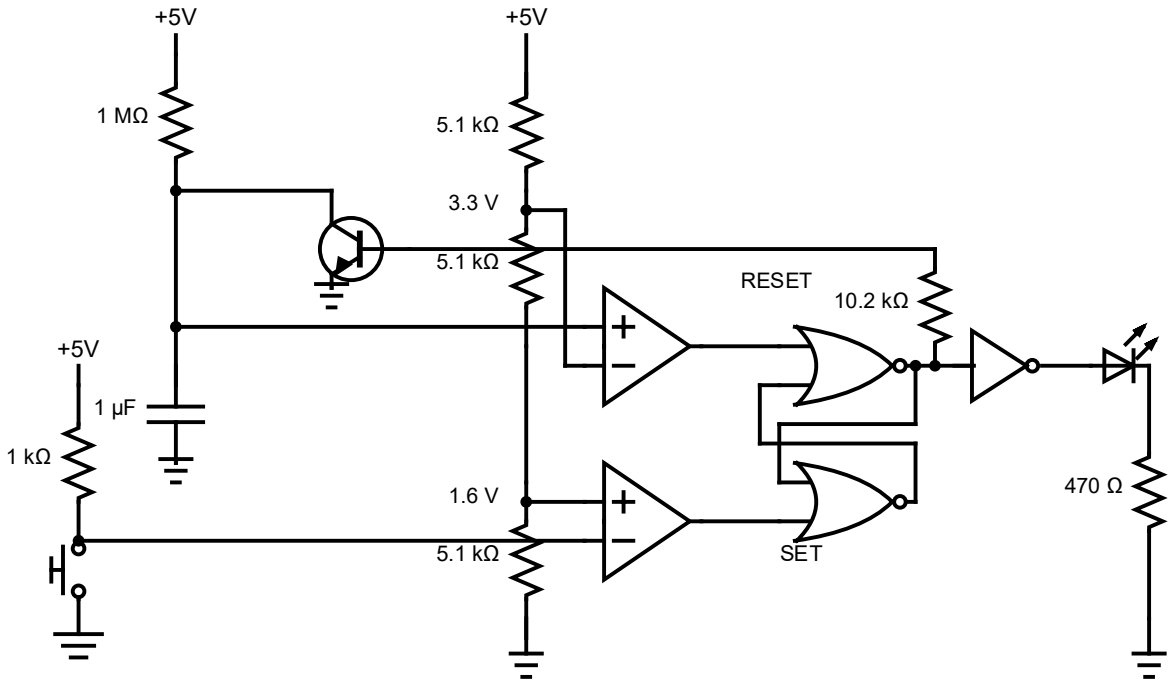



Figure 6. The 555 Time Machine in Monostable Mode

Figure 6 depicts the 555 timer in monostable mode. When in a stable state, both the set and reset comparators are in the low state; 0 V is presented on the inverting input of the set comparator, and 1.6 V is presented on the non-inverting input. On the reset comparator, 3.3 V is presented on the inverting input, and 0 V is presented on the non-inverting input. In this state, the LED is off.

When the button is pressed, the inverting input of the set comparator is pulled to 0 V. This triggers the set pin of the SR Latch, switching on the LED, and switching off the discharge transistor. This causes the capacitor to charge up through the 1 MΩ resistor. This is the non-stable state. The capacitor continues to charge until it reaches 3.3 V. At this point, the reset comparator is high, resetting the SR Latch. This switches the LED back off, and the transistor back on, providing a direct path to ground for the capacitor, instantly discharging the capacitor, bringing the timer back to its stable state. It remains in the stable state until the button is pressed again (see Figure 7).

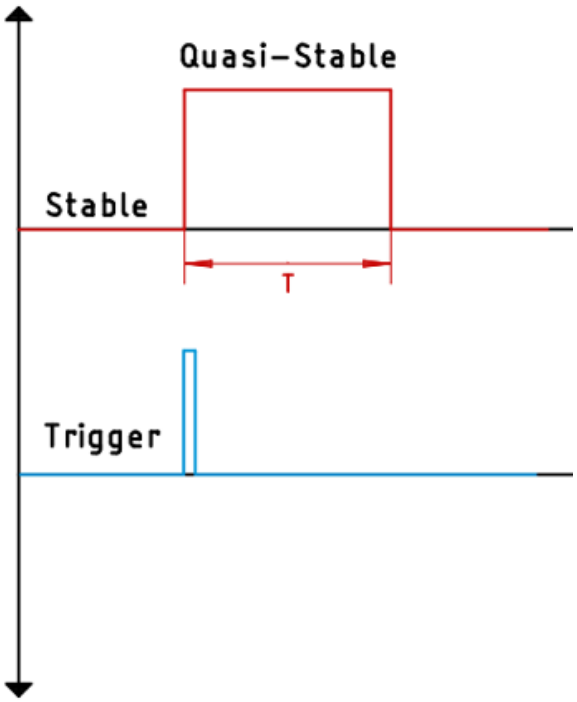
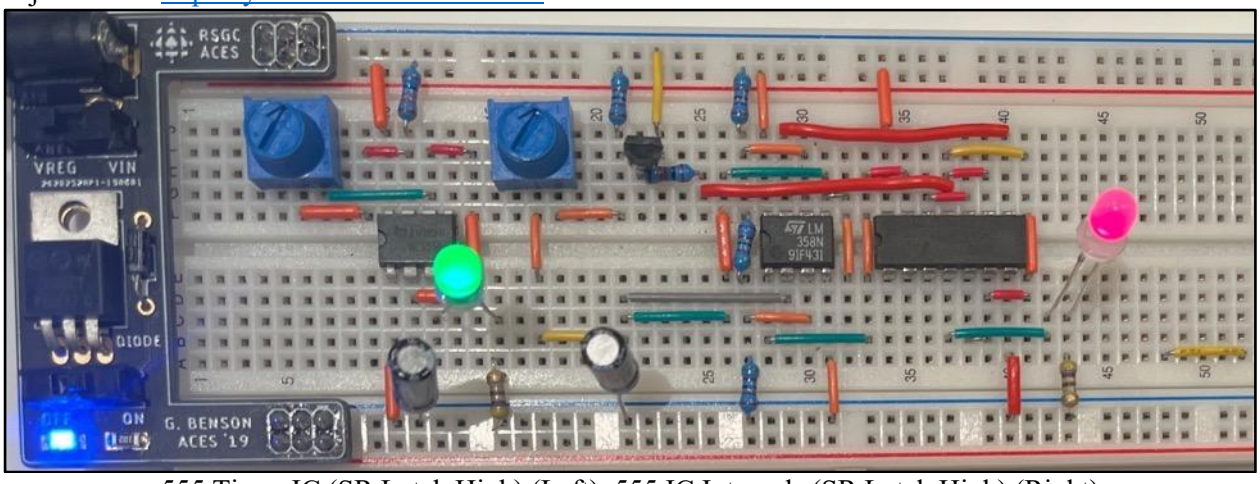


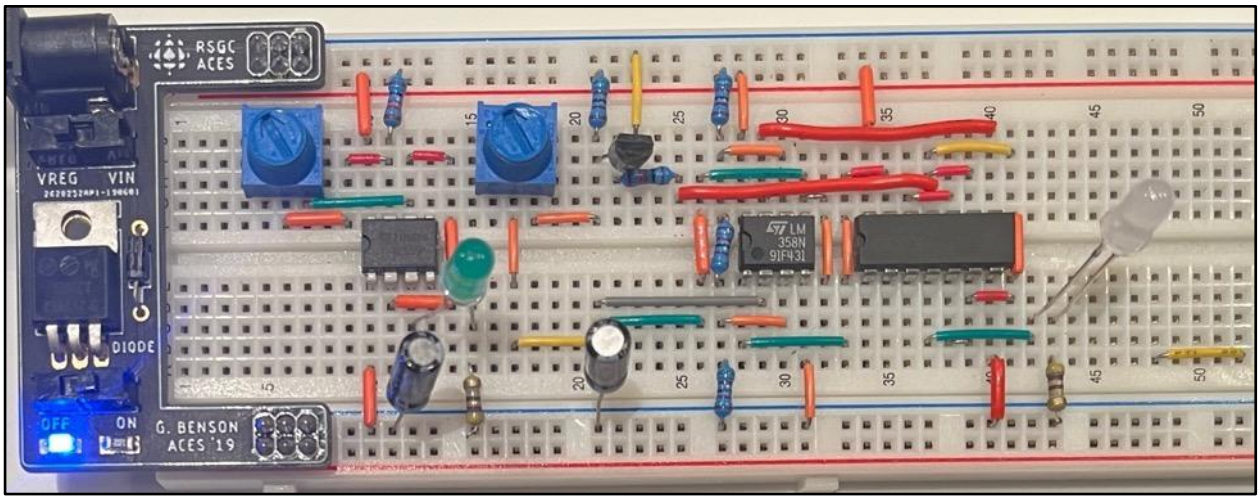

Figure 7. Output in Monostable Mode

## Media

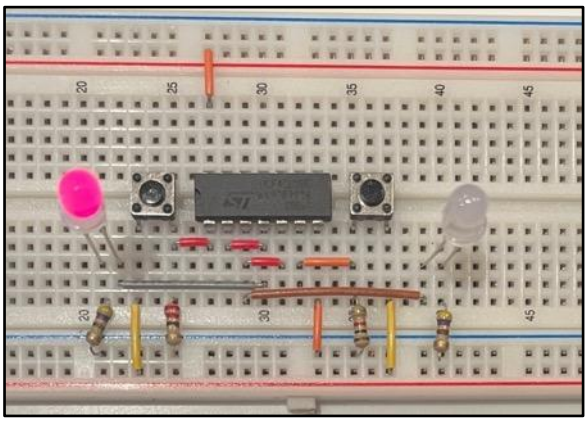
Project video: [https://youtu.be/YrfZDEGoAbw](https://youtu.be/YrfZDEGoAbw)



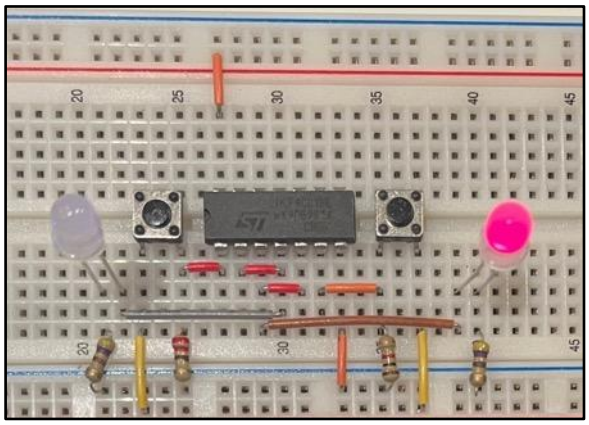

555 Timer IC (SR Latch High) (Left), 555 IC Internals (SR Latch High) (Right)




555 Timer IC (SR Latch Low) (Left), 555 IC Internals (SR Latch Low) (Right)



SR Latch Q High



SR Latch Q Low

## Reflection

One of my favourite things about this course is how thoroughly the writing of a DER can teach you. Before writing my DER, when I looked at the schematic of the internals of the 555, it looked remarkably complicated. Even though in the back of my mind, I knew that, in time, it would become extremely simple, I couldn't even begin to imagine how the circuit worked. Now, after writing this report, I can easily explain how the circuit works. I don't know what it is about DERs. The 555 timer circuit now seems incredibly obvious to me, which is completely different than before the project. As far as time management goes for this project, things went pretty smoothly. Unlike my last project, where I did most of it in one [late] night, I worked on this project over a longer period of time. Ben Eater's videos about the SR Latch, monostable 555 and astable 555 all helped me understand the circuit. This allowed me to write about the circuit in more detail, as I didn't need to research it in tandem. Overall, I think that this has been another successful project, and I think that the 555 will be an essential component in future projects.

# Project 2.3 Multi-Bit Memory (Register)

## Purpose

In addition to demonstrating the data-storing capabilities of the Data Flip-Flop (DFF), the purpose of the register is to serially store 8 bits, making a byte.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI3M/Tasks.html#DFlipFlop

Project schematic: https://crcit.net/c/84875c7560c1491eaa790e1f1c7e833f

## Theory

The DFF is a crucial combinational logic circuit in modern computing, and is used to store the simplest unit of computer memory, a bit. The bit gets its name from the words binary digit, and is simply a stored 1 or a 0. A simple DFF has two inputs and two outputs: a data input, and a clock input, as well as an output and an inverted output. When the clock is high, the output of the DFF is controlled by the data input; the signal presented on the data input corresponds to that of the output (see Figure 1). When the signal goes low, the DFF latches onto its current state, whether high or low, effectively storing the input until the clock goes high again.

Multiple DFFs can be configured in combination to produce a *register*. A register is a multi-bit unit of memory built from DFFs (see Figure 2). In this configuration, the output of each DFF is connected to the next and the clock inputs are wired together. As a result, on each rising edge of the clock signal, the stored data gets shifted one DFF to the right.

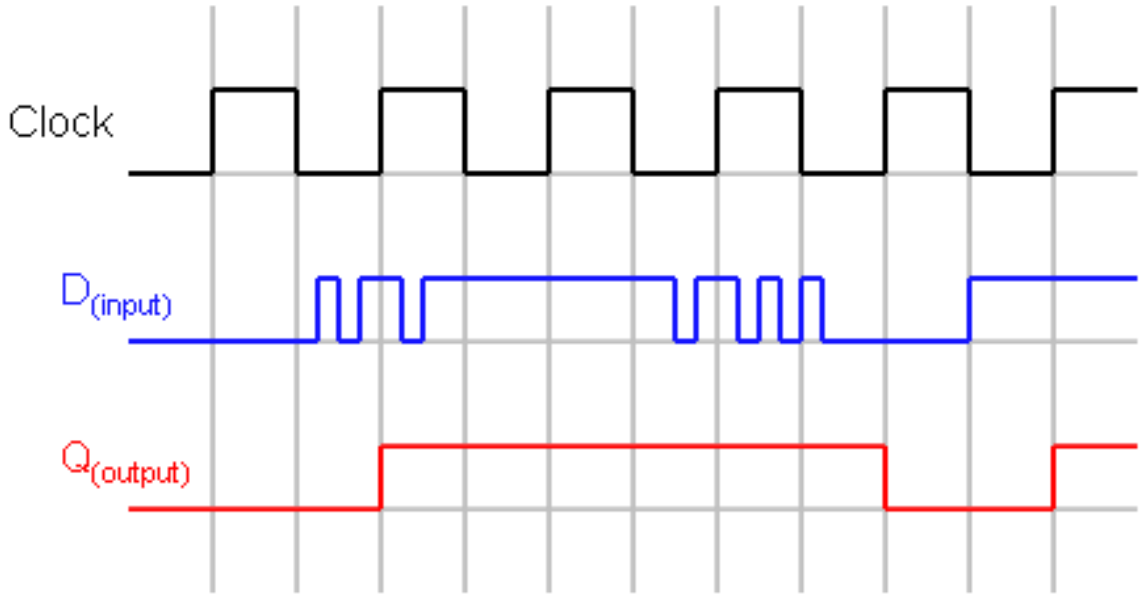



Figure 1. D Flip-Flop Inputs vs Outputs with Clock

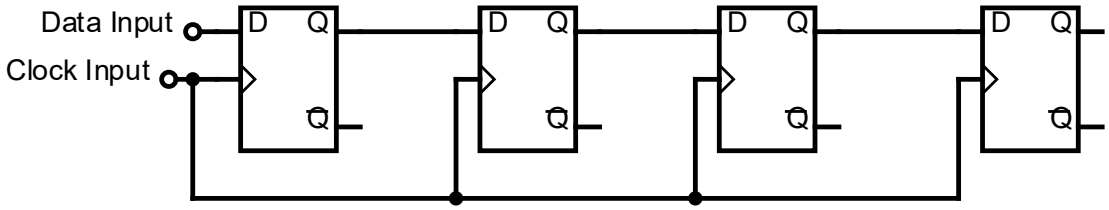



Figure 2. 4 Bit Register

## Procedure

The SN74HC74 IC is a dual D Flip-Flop chip, meaning it contains two separate DFFs. The DFF makes use of the latching properties of the SR Latch (see Project 2.2). Each DFF is capable of storing a single bit of data.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 3 | Momentary Push Button |
| 8 | 5 mm LED |
| 8 | 470 Ω Fixed Resistor |
| 4 | 10 KΩ Fixed Resistor |
| 4 | SN74HC74 Dual DFF IC |
| 1 | NE555 Timer IC |
| 1 | 10 µF Electrolytic Capacitor |
| 1 | 0.1 µF Disk Capacitor |
| 1 | 5V Power Source |
| ~ | Wires |

The DFF is, in principle, a modified *Gated SR Latch* (see Figure 1). A Gated SR Latch is an SR Latch with two additional NAND gates, sharing a common input. This creates a third input: the enable input. In this configuration, the SR Latch only responds to inputs S and R when enable is high.

Much like the non-gated SR Latch, the gated SR Latch still has a flaw; if the latch is enabled, and both the set and reset inputs are high, the latch enters an unpredictable state. A high on both set and reset becomes a low on $\overline{S}$ and $\overline{R}$. With one input low on each NAND gate, both Q and $\overline{Q}$ must be high, meaning the state of the latch is invalid since Q and $\overline{Q}$ must be in opposite states. This problem is solved with a D Flip-Flop (see Figure 2).

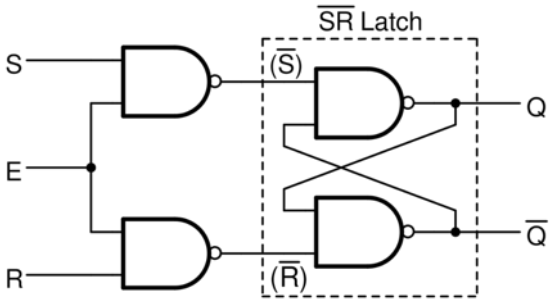



Figure 1. Gated SR Latch

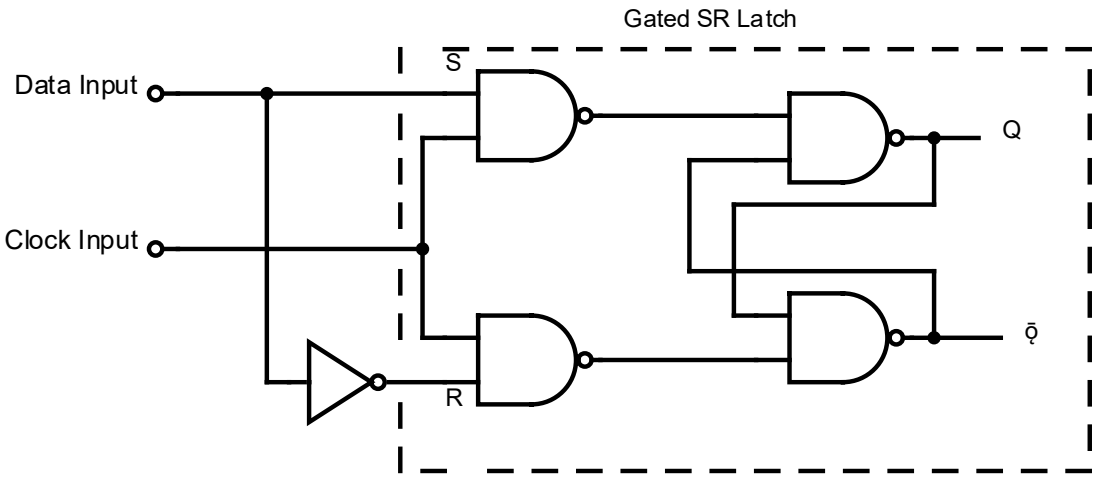



Figure 2. D Flip-Flop

As depicted in Figure 2, the D Flip-Flop only has two inputs: data and clock. The data input is fed directly into the set input of the Gated SR Latch, as well as the reset latch through an inverter gate. This prevents the set and reset inputs from being in the same state. In this configuration, the output of the flip-flop can be switched using just the data input. With the clock input connected to the enable input of the Gated SR Latch, the DFF only toggles to the state of the data input when the clock input is on a rising edge. This effectively causes the DFF to store the state of the data input during the last high input of the clock signal.

Since the DFF is built almost exclusively from logic gates, the circuit can be built using transistors in place of logic gates (see Figure 3). In this configuration, there are no integrated circuits; the logic gates are emulated by transistors in certain configurations.

Figure 3 depicts a NAND gate built from NPN transistors. NPN transistors only allow current to pass through when a high signal is presented on the base of the transistor. This means that the output of the entire gate is pulled high by R1, until a direct path to ground is provided. The only way for this to happen is for both transistors to allow current to pass through, or for both inputs A and B to present a high signal.

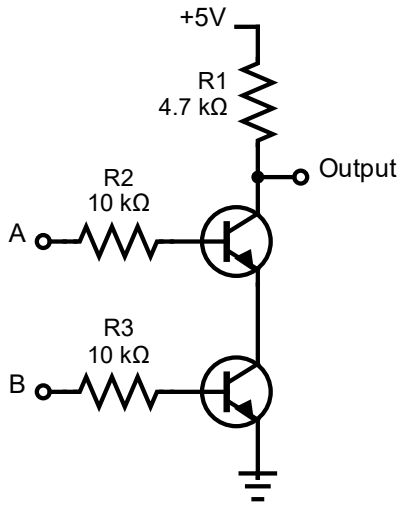



Figure 3. NAND Gate from Transistors

This is the expected behaviour of a NAND gate, according to its truth table (see Figure 4). The inverter gate on the DFF (see Figure 2) can be emulated using the same circuit as the NAND gate, eliminating R3 since the inverter gate is the equivalent to a NAND gate with its inputs tied together.

**Figure 4. NAND Truth Table**

| A | B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

8 DFFs can be wired together to serially store 8 bits, or a byte, in an 8-bit register (see Figure 5). In this configuration, the input of the DFF H is connected to a pushbutton in a pull-down resistor configuration. Each output is wired to an LED to indicate the status of the output, as well as the next DFF, until DFF A is reached, which is simply connected to an LED.

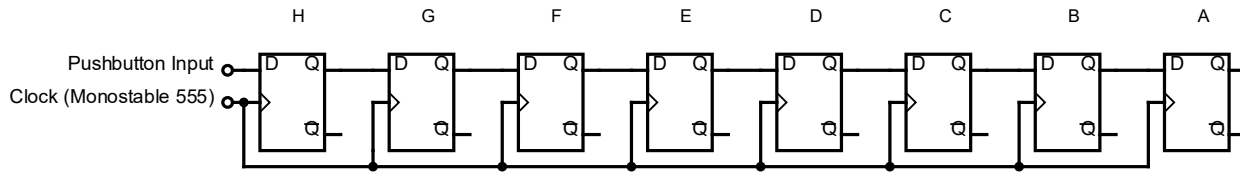



Figure 5. 8 Bit Register

The clock signal is sourced from a 555 timer in monostable mode (see Project 2.2). In this configuration, the timer is configured to debounce the input from a momentary push-button. Figure 5 depicts a serial register. This means that each bit is entered serially, or in sequence. On each rising edge of the clock signal (when the clock button is pressed) the state of the data pin of each DFF is stored. This means that each DFF stores the previous state of the DFF to the left, or in the case of DFF H, the state of the pushbutton. This process can be repeated an unlimited number of times, meaning a register with any number of bits could be built. To store data to the last DFF (A) the data must first be stored to DFF H, and shifted down to A with seven clock cycles.

For simplified building of the 8-bit register, a SN74HC74 IC is employed. This integrated circuit contains two DFFs (see Figure 6), as well as some additional functions. It two active low inputs, clear and preset. As an active low input, for normal operation, these inputs must be pulled high. To have a proper memory-clearing button on the 8-bit register, each clear pin must be wired together, to a common momentary pushbutton. Since it is an active low input, the pushbutton employs a pullup resistor, pulling the clear input low when pressed by the operator.

The SN74HC74 is a 5 V IC, meaning that it will get damaged if the supply voltage exceeds 5 V. A voltage regulator, such as the 7805CT can be used to step down the voltage to 5 V.

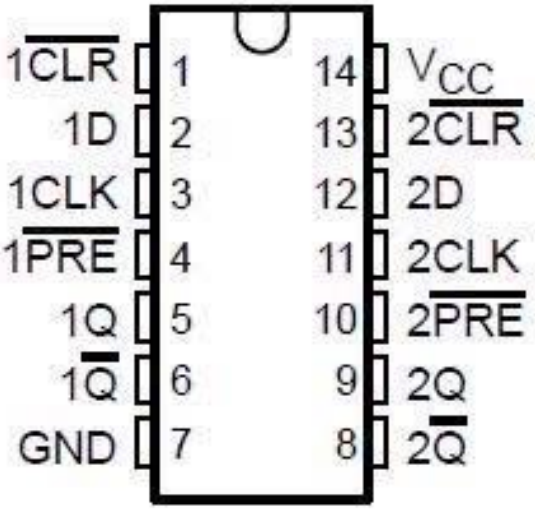
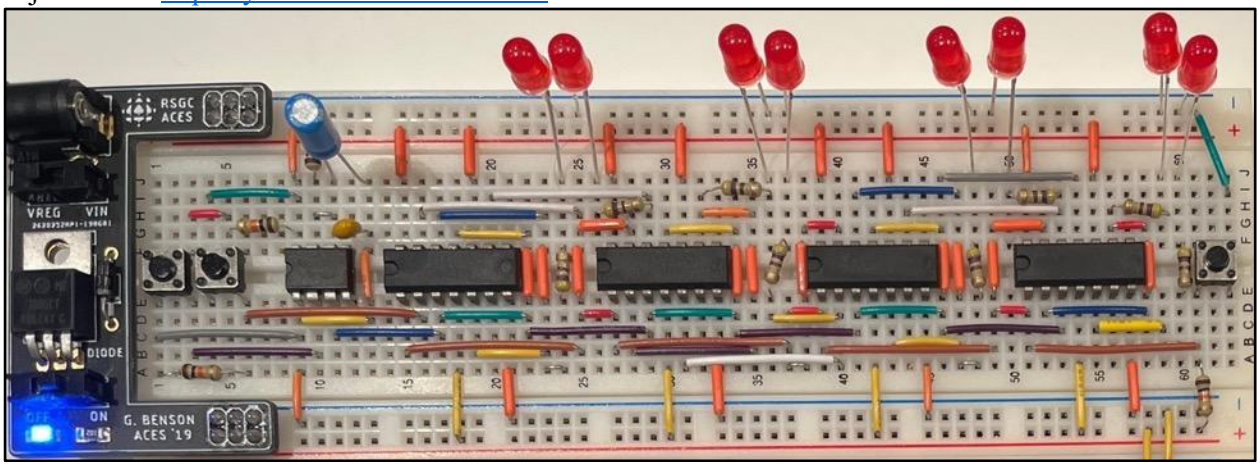


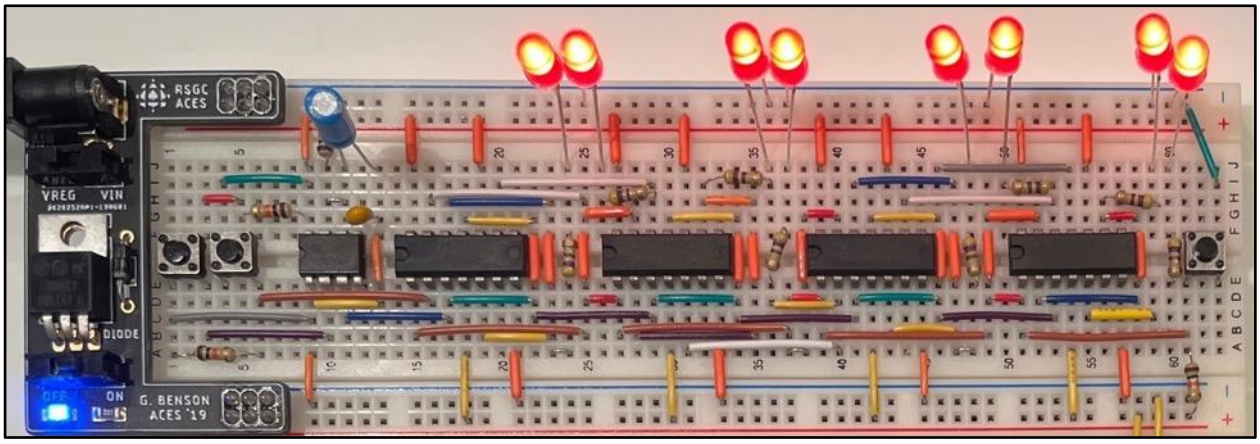

Figure 6. SN74HC74 Pinout

## Media

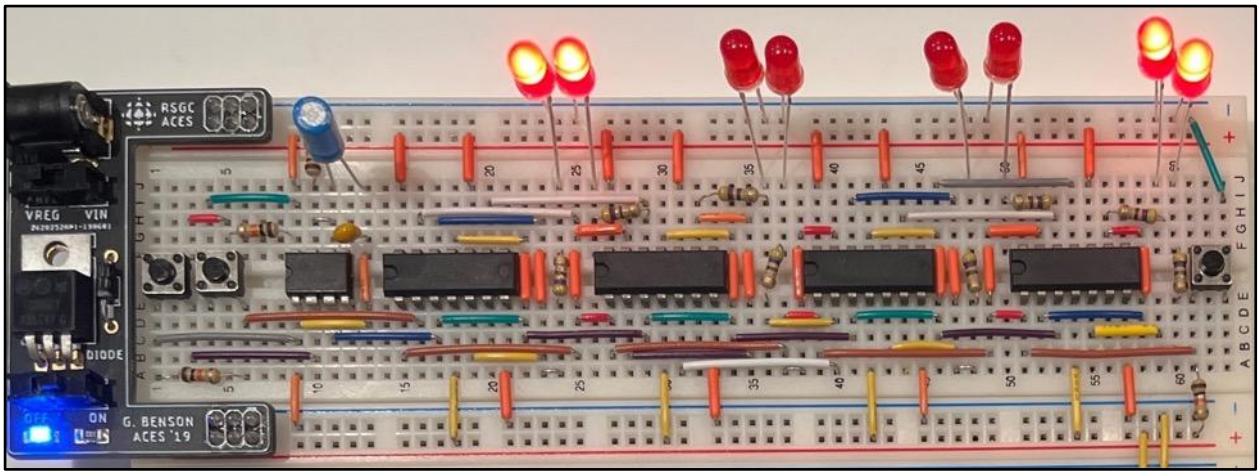
Project video: https://youtu.be/VASVHG98tYk



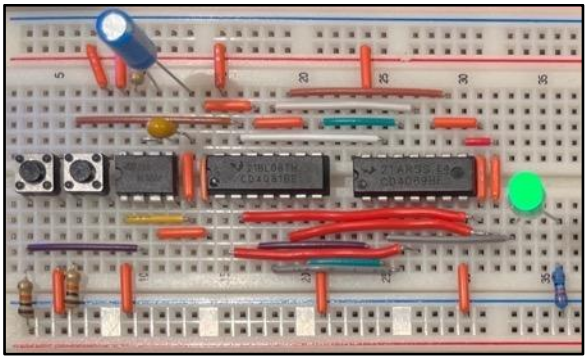

All D Flip-Flops Low (00000000)



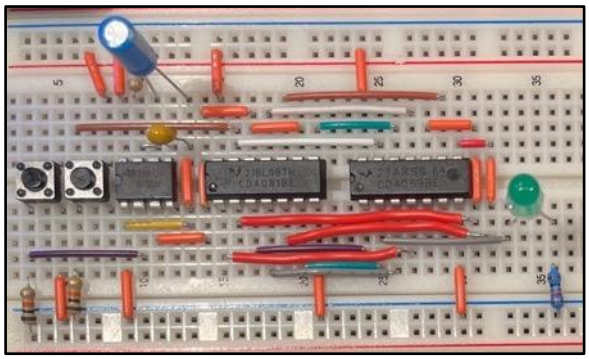

All D Flip-Flops High (11111111)



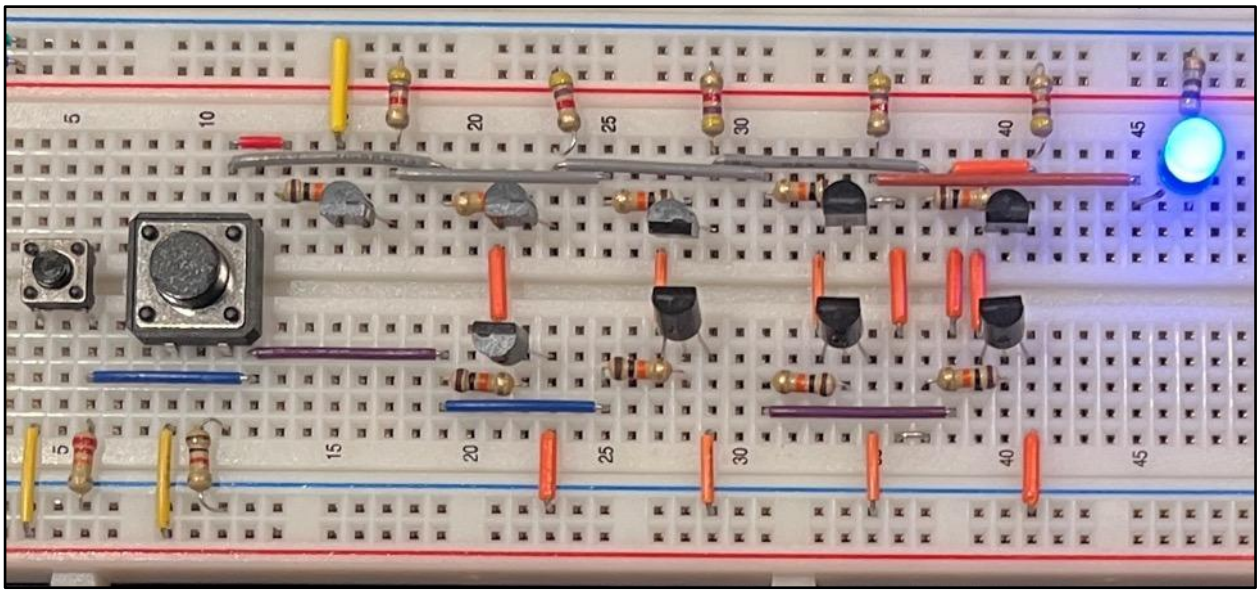

Two D Flip-Flops High, Six Low (11000011)



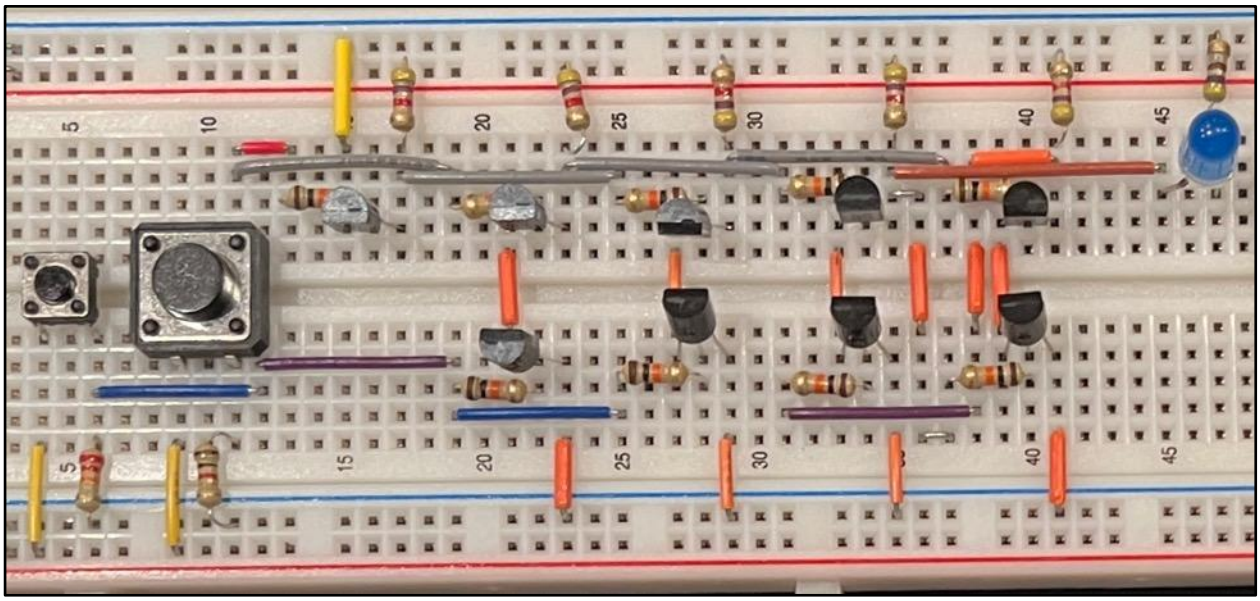
D Flip-Flop (From Logic Gates) Stored 1



D Flip-Flop (From Logic Gates) Stored 0



D Flip-Flop (From Transistors) Stored 1



D Flip-Flop (From Transistors) Stored 0

## Reflection

Out of all the projects (in both grades 10 and 11) I think that this is definitely my favourite one. I wish that I had started sooner, but instead I spent time working on my ISP since I thought it was much more interesting. The reason that I like this project is that it seems much less theoretical than the others; things like the NAND Gate Oscillator, or 2-bit Magnitude Comparator, or R/2R DAC are all cool builds, but as far as practicality in modern computing goes, it is somewhat difficult to find a connection. For this project, it is pretty clear that registers are an important component, and DFFs are, from what I can tell, a necessary component.

I also find it cool that I was able to build a storage device from just transistors. It's one thing to have built a complex circuit from just chips, but it's almost surreal to have built it from transistors. When it's a chip that's being used, it is almost like a black box, the chip could be doing all the heavy lifting. But when it's built with transistors, not only is it just raw components giving you the final result, but it's also another layer of thinking. When I built a DFF out of logic gates, it was as simple as connecting all the gates together according to the schematic. But when it came to building a DFF from transistors, I actually had to sit down, think, and then build. I must admit, sometimes, I'm on the wrong side of the ACE-berg graphic, but this project put me back on track for building to be just the tip of the iceberg (with thinking being the rest). Overall, this was a great project, and now that it's done, I have time to finish soldering my ISP so I can start 3D printing the case.

# Project 2.4 Binary Game. Part 1

## Purpose

In addition to exhibiting the I/O expanding properties of shift registers, the purpose of the Binary Game. Part 1 is to echo the states of a rocker DIP switch bank on the Morland Bar Graph.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI3M/Tasks.html#BinaryGame1

Project schematic: https://crcit.net/c/6d1e62c403f8474da6af594dffa33452

## Theory

Shift registers can be used to extend the number I/O of a *microcontroller unit* (MCU). An MCU is a small integrated circuit for use within an embedded system. In the Arduino IDE, there are two ways to send the desired data to the shift register: using software (`shiftOut()`) and using hardware (Serial Peripheral Interface). The software implementation, `shiftOut()`, is able to send a byte of data one bit at a time. The shiftOut function has four parameters: dataPin, clockPin, bitOrder, and value. The first two parameters pertain to the main input pins on the shift register (see Project 2.3), data and clock. They both accept "int" data types, or signed 16-bit integers that indicate the associated I/O pin on the MCU. The bitOrder parameter indicates the order in which the bits should be shifted out, and accepts a 0 or a 1. 0 tells the `shiftOut()` function to send the *most significant bit* (MSB) first, while 1 sends the *least significant bit* (LSB) first. The MSB is leftmost bit, while the LSB is the rightmost bit. The Arduino IDE contains the built-in constants MSBFIRST, which has a value of 0, and LSBFIRST, which has a value of 1. The value parameter accepts a single byte, or any decimal value between 0 and 255.

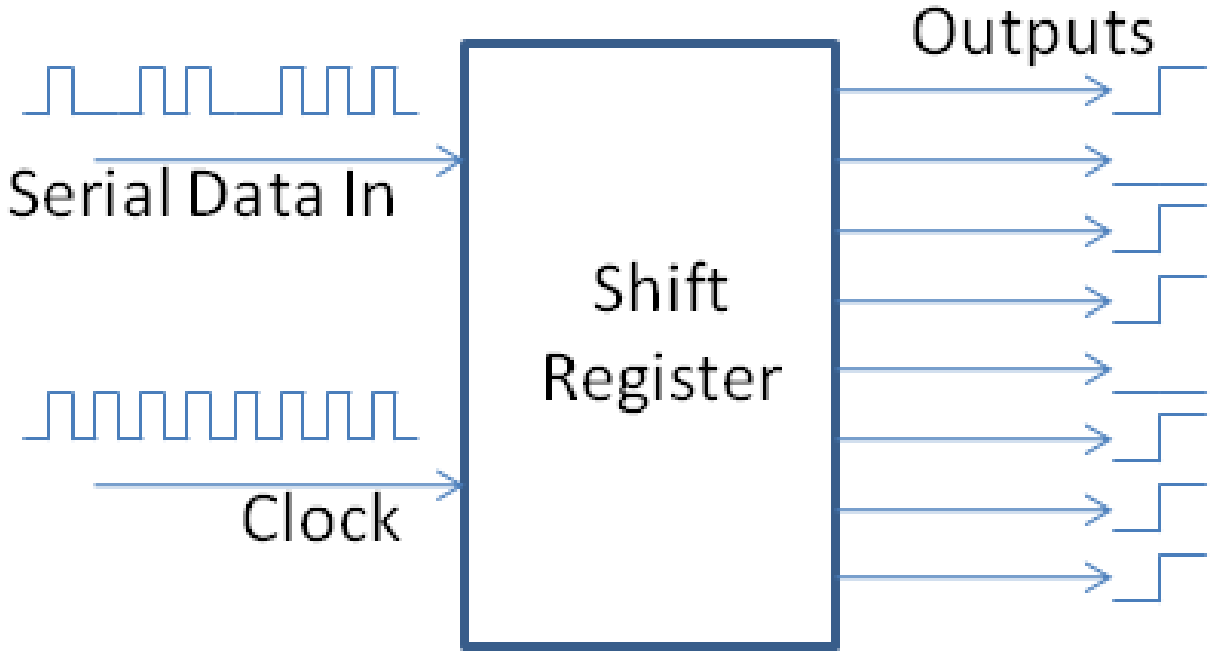



Figure 1. Arduino shiftOut Timing

When the bits are being shifted out, the data pin goes to the state pertaining to the current bit to achieve the desired value (see Figure 1). To move to the next bit, the clock pin is taken high, then low, to indicate to the shift register that the next bit is ready. The hardware implementation of `shiftOut()`, serial peripheral interface (SPI) is much faster, but only works on specific pins. Additionally, SPI is able to send and receive data simultaneously.

## Procedure

The Binary Game. Part 1 makes use of an Arduino Nano to echo the state of a rocker DIP switch bank on the Morland Bargraph, and the serial monitor. It accomplishes this through reading each individual switch, and assembling them into a decimal number with *binary-weighted values*. This means that the leftmost switch represents the MSB, in this case decimal 128, whereas the rightmost switch represents the LSB, which is decimal 1.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | Rocker DIP Switch Bank |
| 8 | 10 KΩ Fixed Resistor |
| 1 | Morland Bargraph |
| 1 | Arduino Nano |
| ~ | Wires |

The DIP switch is assembled into a decimal number using *bitwise operators* (see Figure 1). Bitwise operators are used to perform operations on individual bits of binary numbers. These operators work at the binary level and manipulate the binary representation of data. The Binary Game. Part 1 makes use of the bitwise OR operator, as well as the bitwise shift left operator.

| Figure 1. Bitwise Operators | |
|---|---|
| **Operator** | **Description** |
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise XOR |
| ~ | Bitwise NOT |
| << | Bitwise shift left |
| >> | Bitwise shift right |

The Arduino Nano first uses the `digitalRead()` function to read the status of each switch, and uses the bitwise shift left operator to shift each bit left, with the number of shifts corresponding with the significance of the bit. The LSB is not shifted, the second LSB is shifted once, up until the MSB, which is shifted 7 times to the left. A bitwise shift left is mathematically equivalent to multiplying by $2^n$ where n is the number of bitwise shifts left. Bit manipulation is more efficient than multiplying, since MCUs have built in circuitry, for fast bit manipulation.

To combine each bit into a single byte, the bitwise OR operator is used. This compares each binary number, bit by bit, applying the OR truth table to it (see Figure 2), returning a single 8-bit binary number. This allows the value stored in each bit to be passed through to the final byte (see Figure 3).

| Figure 2. OR Truth Table | | |
|---|---|---|
| **A** | **B** | **Output** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| Bit | Figure 3. Sample Byte Assembly (After Shifting) | | | | | | | | Decimal |
|---|---|---|---|---|---|---|---|---|---|
| **MSB** | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 128 |
| **7** | 0 | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **6** | 0 | 0 | **0** | 0 | 0 | 0 | 0 | 0 | 0 |
| **5** | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 16 |
| **4** | 0 | 0 | 0 | 0 | **0** | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 0 | 4 |
| **2** | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 2 |
| **LSB** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** | 0 |
| **RESULT** | **1** | **0** | **0** | **1** | **0** | **1** | **1** | **0** | **150** |

Once the Arduino assembles the byte and stores it in RAM as an unsigned 8-bit integer, it reads it out to the serial monitor, as an 8-bit binary number. It also uses `shiftOut()` (see Theory section) to echo the number on the Morland Bargraph. First, the latch is set to low, or inactive, so that a new value can be programmed into the shift register. Once the latch is disabled, the `shiftOut()` function programs the stored value into the shift register, and enables the latch again for the value to update and remain on the bargraph. The shift register used is the 74HC595 (see Figure 4).

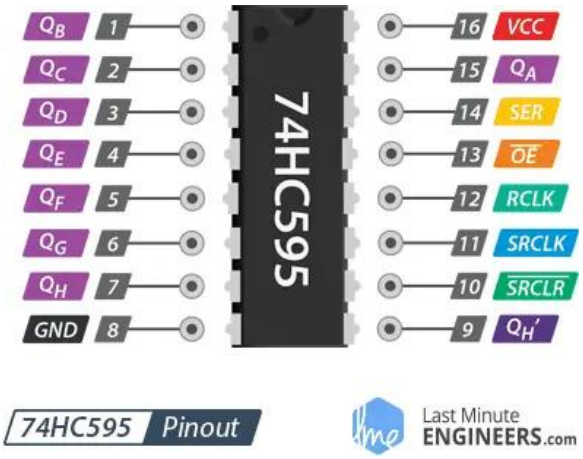



Figure 4. 74HC595 Pinout

## Code

```
// PROJECT   : Binary Game. Part 1
// AUTHOR    : R. Jamal
// PURPOSE   : To echo the status of a DIP rocker on the Morland Bargraph
// COURSE    : ICS3U-E
// DATE      : 15 11 2023
// MCU       : 328P (Nano)
// STATUS    : Working
// REFERENCE : http://darcy.rsgc.on.ca/ACES/TEI3M/Tasks.html#BinaryGame1
// NOTES     : Created in 1.8.19
#define CLOCKPIN 10           //Define I/O pin for clock signal
#define DATAPIN 12            //Define I/O pin for data
#define LATCHPIN 11           //Define I/O pin for latch
uint8_t value;                //Create variable to store state of rocker DIP
uint8_t value2;               //Create variable to detect when state changes
void setup() {
  Serial.begin(9600);         //Establish serial communication and set Baud rate
  pinMode(CLOCKPIN, OUTPUT);  //Define CLOCKPIN as output
  pinMode(DATAPIN, OUTPUT);   //Define DATAPIN as output
  pinMode(LATCHPIN, OUTPUT);  //Define LATCHPIN as output
}
void loop() {
  while (value == value2) {   //Monitor DIP rocker state
    value = digitalRead(9) << 7 | digitalRead(8) << 6 | digitalRead(7) << 5 |
            digitalRead(6) << 4 | digitalRead(5) << 3 | digitalRead(4) << 2 |
            digitalRead(3) << 1 | digitalRead(2);  //Print and assemble byte in DIP
  }
  Serial.println(value, BIN);                      //Print "value" in binary
  digitalWrite(LATCHPIN, LOW);                     //Disable latch
  shiftOut(DATAPIN, CLOCKPIN, MSBFIRST, value);    //Program value to Bargraph
  digitalWrite(LATCHPIN, HIGH);                    //Enable latch to make value live
  value2 = value;                                  //Set variable to detect changes
}
```

## Binary Game. Part 1.5

### Purpose

The purpose of the Binary Game. Part 1.5 is to provide a decimal-to-binary conversion game on the Arduino IDE serial monitor interface with a DIP rocker switch as input.

### Procedure

Version 1.5 of the Binary Game is similar to version 1, with some differences in both functionality and technicality. Unlike version 1, version 1.5 generates a random 8-bit number, and compares it to the value presented on the rocker DIP switch. It then waits for the user to enter the presented number, and gives the user a point when the number is entered.

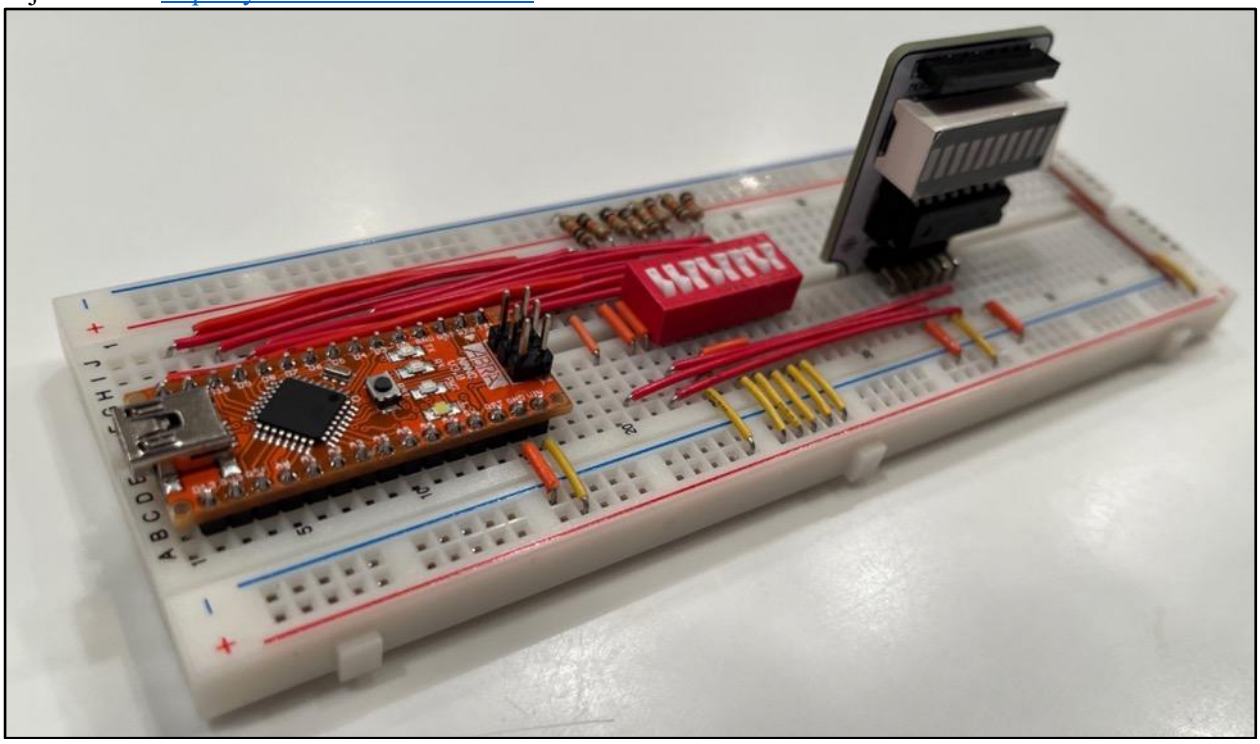
This is accomplished with additional lines printed in the serial monitor to communicate with the user the desired number, as well as their earned points. A modified while loop continuously monitors the input and updates the bargraph until the correct number is entered. After this, the point counter is incremented.

### Code

```
// PROJECT   : Binary Game. Part 1.5
// AUTHOR    : R. Jamal
// PURPOSE   : To generate random numbers with DIP input for BIN conversion game
// COURSE    : ICS3U-E
// DATE      : 17 11 2023
// MCU       : 328P (Nano)
// STATUS    : Working
// REFERENCE : http://darcy.rsgc.on.ca/ACES/TEI3M/Tasks.html#BinaryGame1
// NOTES     : Created in 1.8.19
#define CLOCKPIN 10           //Define I/O pin for clock signal
#define DATAPIN 12            //Define I/O pin for data
#define LATCHPIN 11           //Define I/O pin for latch
uint8_t value;                //Create variable to store state of rocker DIP
uint8_t ran;                  //Create variable to store pseudo-random number
uint16_t points = 0;          //Create variable to store number of points
void setup() {
  Serial.begin(9600);         //Establish serial communication and set Baud rate
  pinMode(CLOCKPIN, OUTPUT);  //Define CLOCKPIN as output
  pinMode(DATAPIN, OUTPUT);   //Define DATAPIN as output
  pinMode(LATCHPIN, OUTPUT);  //Define LATCHPIN as output
  randomSeed(analogRead(A0));  //Initialize number generator seed with random seed
}
void loop() {
  ran = random(0, 255);       //Set ran to a random number
  Serial.print("Number: ");   //Print "Number:" followed by a space
  Serial.print(ran);          //Print contents of ran on the same line
  Serial.print("\t \t");      //Print 2 tabs
  Serial.print("Points: ");   //Print "Points:" followed by a space
  Serial.println(points);     //Print points and go to next line
  while (ran != value) {
    value = digitalRead(9) << 7 | digitalRead(8) << 6 | digitalRead(7) << 5 |
            digitalRead(6) << 4 | digitalRead(5) << 3 | digitalRead(4) << 2 |
            digitalRead(3) << 1 | digitalRead(2);  //Print and assemble byte in DIP
    digitalWrite(LATCHPIN, LOW);                   //Disable latch
    shiftOut(DATAPIN, CLOCKPIN, MSBFIRST, value);  //Program value to Bargraph
    digitalWrite(LATCHPIN, HIGH);                  //Enable latch to make value live
  }
  points++;                                        //Increment points by 1
}
```

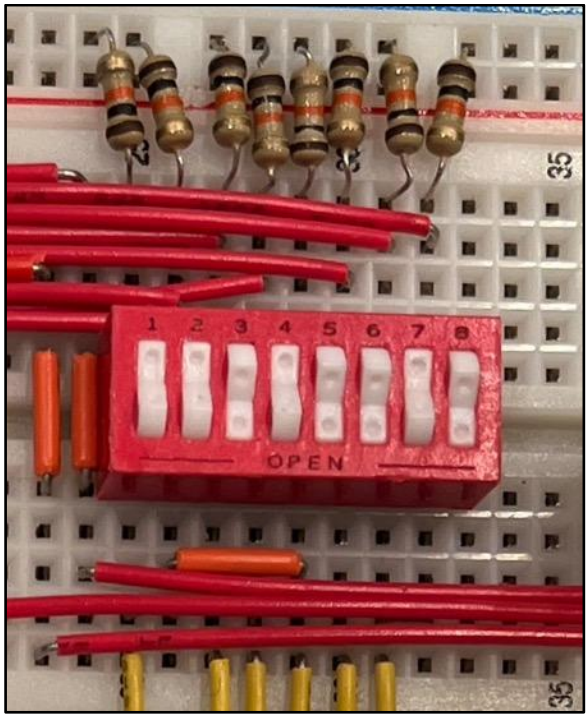
## Media

Project video: https://youtu.be/zNxUtwbOslM

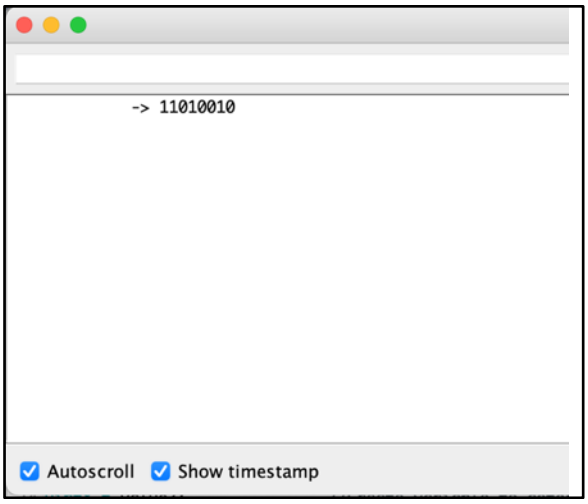


Binary Game. Part 1 Entire Circuit

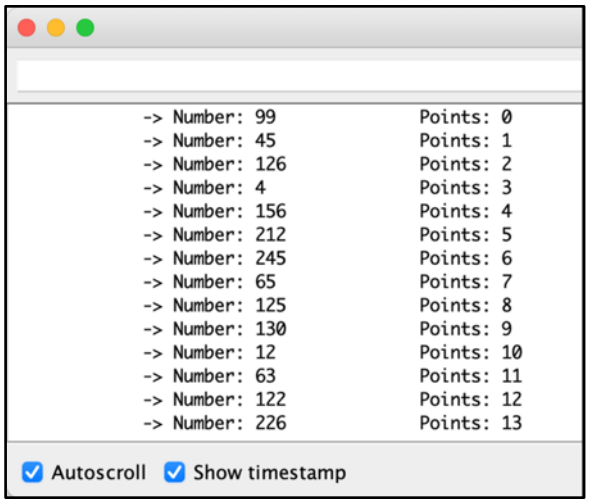

Morland Bargraph: Binary Number 11010010




DIP Rocker: Binary Number 11010010




Part 1: Serial Monitor, Binary Number 11010010




Part 1.5: Serial Monitor, 13 points played

## Reflection

I think that this was a pretty good introduction to software for me. While there were some basic things that I already knew (like the read and write functions) I had never even heard of shiftOut before this project. Before this project, the idea of using a shift register to enhance I/O was just that far out idea that I knew was possible somehow, but had no real idea how to implement it. While SPI did intimidate me more than a little bit, shiftOut was pretty straightforward to use, which is good. One software lesson that I learned early on (but still a bit too late) was to do the comments in the Arduino IDE, not the table. There were multiple times that I had to change a line, but since my comments were put in after changing copy-pasting my code, I had to redo most of my comments. Eventually I realized that I could just copy the code from the table, into the IDE, change it and then bring it back, but it still did cost me a fair amount of time.

I think that the ability to use a shift register to extend I/O opens up a lot of doors. To be completely honest (aside from PWM and analog capabilities) I don't see the real point of the ATmega328P anymore, other than convenience. I feel like I would almost always just be able to use an ATtiny85 with shift registers to get as many I/O pins as I need. I wish I knew about shift registers when I started my ISP since I ended up basing the whole design on the number of available I/O pins. I guess that's the thing with ISPs, though; no matter what, a couple months down the road, there will always be something you wish you knew at the start, but you didn't. Overall, this has been a great project, and I'm glad I'm done since I had better get to finishing off the ISP so that I have a buffer period, in case something goes wrong.

# Project 2.5.3 Short ISP: Bike Computer

## Purpose

The purpose of my Short ISP, which is a bike computer, is to display the speed and distance of a bicycle on seven-segment displays, based on readings from a reed switch on the fork of the bike

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI3M/2324/ISPs.html#logs

Project proposal: http://darcy.rsgc.on.ca/ACES/TEI3M/2324/images/RohanShort.png

## Theory

A reed switch is a magnetic field-sensing switch. It operates on the same principles as a normal switch (comes in either normally open and normally closed varieties) but instead of being operated by a user, its state depends on the presence of a magnetic field. Inside a normally open reed switch (which is made of glass) are two metal contacts, connected to the only two leads of the component (in most cases). These contacts are constructed from a ferromagnetic material, meaning that they are susceptible to being influenced by a magnetic field. In practice, this means that when a magnetic field is present, the two contacts touch, allowing current to flow between the two leads, unimpeded, just like a switch or pushbutton (see Figure 1).

This property allows them to be used in a variety of applications, including the position of circular objects, including wheels. By placing a magnet at a certain position of the wheel, and a reed switch at a fixed point that the magnet will pass, on every revolution, the reed switch will go high. This means that the generated pulses allow the revolutions per minute (RPM), and total revolutions to be calculated with the addition of an MCU.

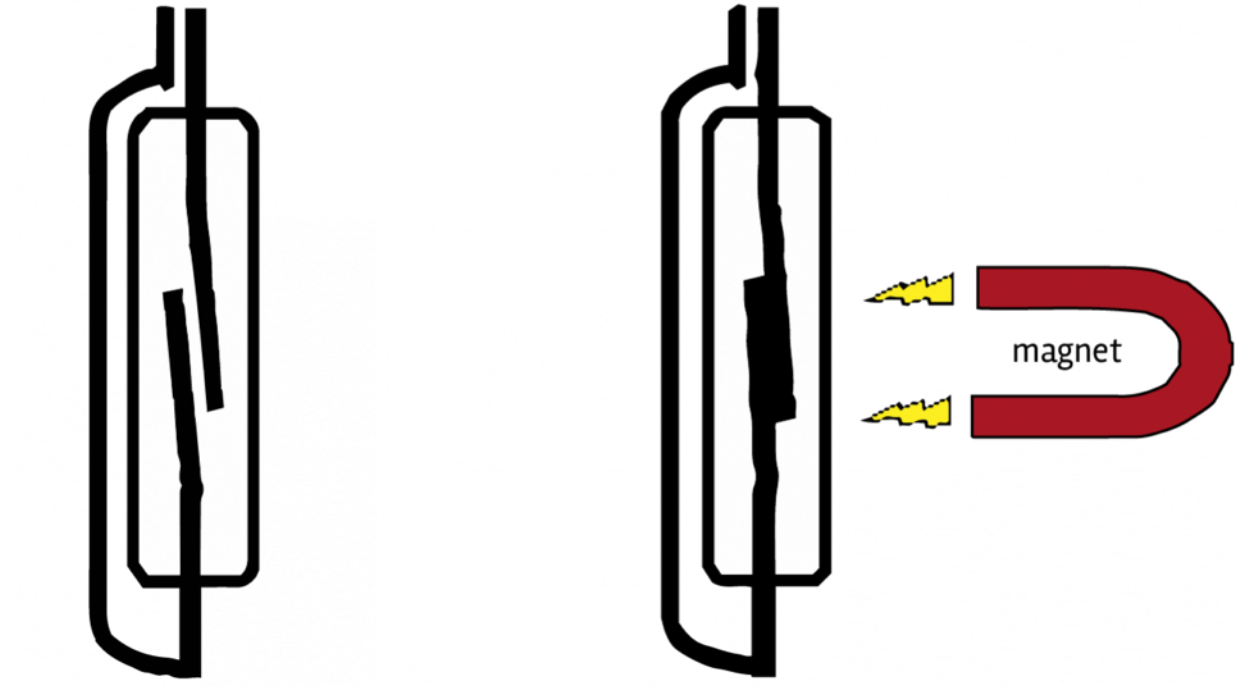



Figure 1. Reed Switch Normally Open with and without Presence of Magnetic Field

## Procedure

The bike computer reads out either the speed of the bike, or the distance the bike has travelled since the device has powered on, based on readings from a reed switch (see Theory section). The computer contains a built-in ATmega328P, which takes input from the reed switch, processes it, passes the appropriate binary number to four CD4511 chips. These decode the binary number, into a human-readable output presented on four seven-segment displays (see Figure 1).

The MCU is able to measure the time between the pulses generated by the reed switch, and calculate the speed of the bicycle from its wheel circumference, using the following mathematical relationship: $S = 3.6 \times \frac{C}{\Delta T}$ where S is speed in KPH, C is the circumference of the wheel in mm, and ΔT is the time it takes a revolution to complete in milliseconds. For example, if a revolution took 1 full second to compete, and the wheel had a circumference of 2155 mm, the speed would be: $3.6 \times \frac{2155\ mm}{1000\ ms} \approx 7.8$ km/h. The distance is measured by multiplying the revolutions completed, with the circumference of the wheel, in km.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 4 | Seven-Segment Display |
| 2 | SPDT Slide Switch |
| 29 | 330 Ω Fixed Resistor |
| 1 | 10 KΩ Fixed Resistor |
| 2 | 2 x 3 Male Header Pin |
| 2 | 1 x 16 Long Male Header Pin |
| 4 | 1 x 8 Female Header Pin |
| 1 | L7805 Linear Voltage Regulator |
| 2 | 0.1 µF Ceramic Capacitor |
| 4 | CMOS CD4511 IC |
| 1 | ATmega328P DIP MCU |
| 1 | 16 MHz Crystal Oscillator |
| 2 | 22 pF Ceramic Capacitor |
| 2 | Terminal Block |
| 6 | 16-Pin DIP Chipseats |
| 1 | 9 V Battery |
| 2 | Perfboard 25 x 24 |
| 1 | DC-In Barrel Jack |
| 1 | Battery Snap |
| ~ | Wires |
| ~ | Solder |

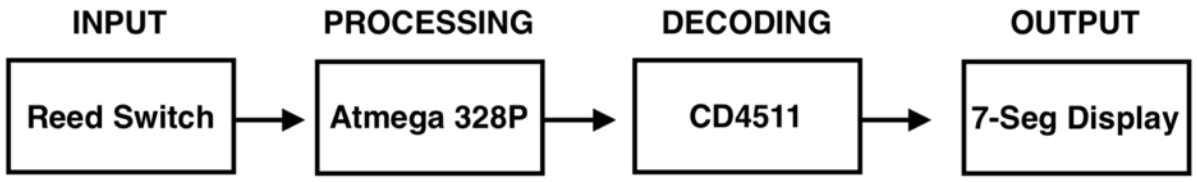



Figure 1. Inputs and Outputs

The device's I/O pins are occupied by a reed switch for input from the bicycle, a user-operated SPDT slide switch to toggle between speed and distance, and 16 pins are occupied by CD4511 chips. The ATmega328P outputs its final number as 4 BCD digits, with the total value multiplied by 10. This is because the second rightmost digit has its decimal place permanently active, so the software must always correct to 1 decimal place. Due to the large footprint of the device and the low amount of space on bike handlebars, the CD4511s, ATmega328P with its supporting hardware, and the 9 V battery are integrated inside a 3D-printed case, connected via header pins to a separate board which contains only the human interface devices (HIDs).

The ATmega328P is the same MCU used in many Arduino microcontrollers, including the Nano and Uno series. When the standard DIP package is used without the addition of the Arduino board, there are a few supporting components that need to be added.

One of the most crucial components is the 16 MHz crystal, with its dual 22 pF "shoulder" capacitors (see Figure 2). By connecting it to pins 9 and 10 (see Figure 3), with the two capacitors between each lead and ground, it produces a square wave at 16 MHz, with a tolerance of ±0.003%, compared to the built-in 8 MHz clock signal's tolerance of ±10%, allowing for increased timing precision.

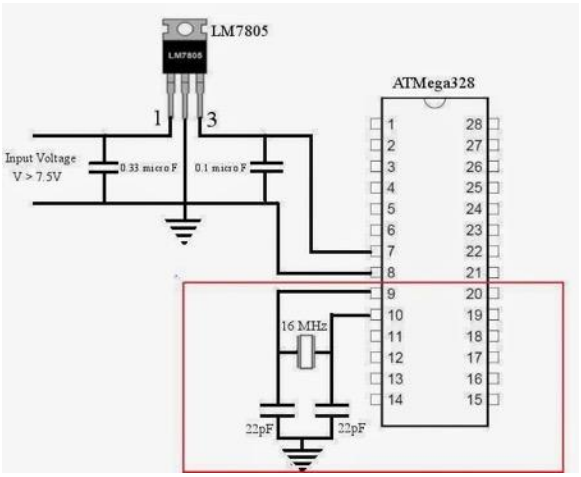



Figure 2. ATmega328P Pinout

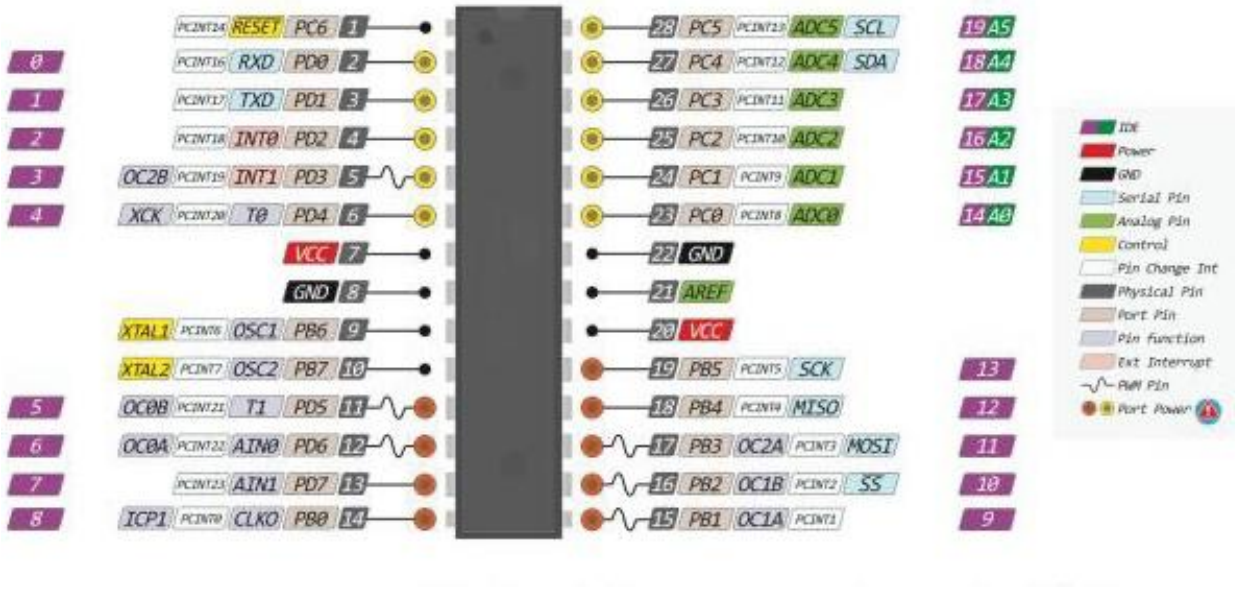



Figure 3. ATmega328P Pinout with Arduino Functions

Figure 2 depicts the other additional component when using an offboard ATmega328P: a voltage regulator. The Arduino contains an onboard voltage regulator, which steps the voltage down to 5 V to prevent damage to the integrated ATmega328P. The output from the voltage regulator is connected to pin 7 of the MCU, and the common ground signal is connected to pin 8 (see Figure 3).

The bike speedometer uses an LM7805 voltage regulator (see Figure 2). This type of regulator is called a *linear voltage regulator*. A linear voltage regulator is a device that maintains a steady output voltage. While they are extremely cheap and versatile, one of their major downsides is their inefficiency. Since they use a negative feedback loop combined with a metal-oxide semiconductor field effect transistor (MOSFET), they are extremely inefficient, effectively dissipating excess voltage as heat. The LM7805 is capable of converting any voltage in excess of 7.5 V to a constant 5 V. This allows the device to be powered by a 9 V battery, despite the fact that it operates at 5 V.

Finally, a 6-pin in-system programmer (ISP) header (see Figure 4) must be added to the circuit in order to program the ATmega328P. On an Arduino board, there is an MCU that converts USB signals to ISP signals. For a standalone ATmega328P, a dedicated in-system programmer must be used. The in-system programmer connects to both power rails, as well as MISO (D12), SCK (D13), reset, and MOSI (D11) (see Figure 3). With the addition of these three components, the standalone ATmega328P has similar functionality to an Arduino board, and can be interfaced with the Arduino IDE.

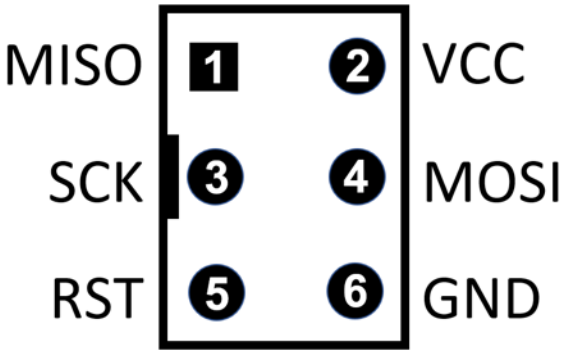



Figure 4. 6-Pin ISP Header Pinout

The code of the bike computer contains 5 variables (see Code section). There are 4 data types used (see Figure 5). A float is used only for kph, the variable that stores the speed of the bike, as a decimal. 64-bit unsigned integers are used for variables that must store the value of `millis()`, a function that returns the running time in milliseconds.

**Figure 5. Arduino Data Types**

| Type | Bytes | Max | Precision |
|---|---|---|---|
| `Float` | 4 | $\sim 2^{128}$ | Decimal |
| `Uint64_t` | 8 | $2^{64}$-1 | Integer |
| `Uint32_t` | 4 | $2^{32}$-1 | Integer |
| `Uint16_t` | 2 | $2^{16}$-1 | Integer |

At the start of the `loop()` function is a while loop, used for timing. The part of the code that doesn't interface with the seven-segment displays is written in the loop. The condition of the loop is that `millis()` is less than `tLoop + 1000`, where `tLoop` was set to `millis()` at the start. This causes the loop to run until more than a second has elapsed, at which point it will execute the code below, which is the reading out of the speed or distance (unless the speed is 0, in which case it updates right away).

Inside the timing loop, the program waits for the magnet to pass the reed switch (see Figure 6), then starts a timer by setting `sTime` to `millis()`. When the wheel comes around again, the program adds one to the total revolutions, and calculates the speed with the equation $S = 3.6 \times \frac{C}{\Delta T}$ where S is speed in KPH, C is the circumference of the wheel in mm, and ΔT is the difference between `sTime` and `millis()`; if, however, the wheel took more than 2 seconds to complete a revolution, the speed is automatically overridden to 0, since the actual speed is very close to 0, and within the override, the speed is read out; the override overrides both the speed, and the timing loop. If the override does not trip, and the timing loop expires (1 second has gone by), the speed or distance is read out.

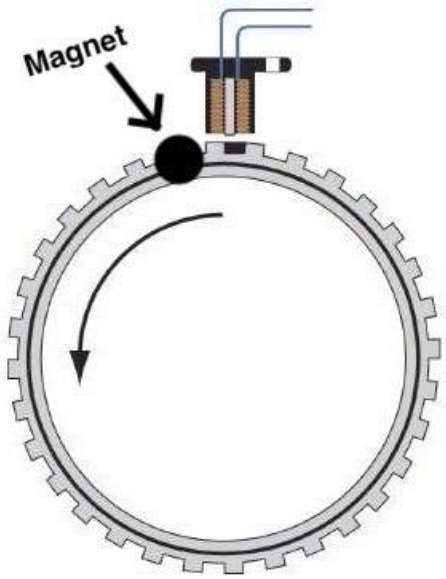



Figure 6. When Rev Timer Starts

There are two additional functions within the code: `lightDisplay()`, and `readOut()`. The latter builds upon the former. Both functions are used to simplify the software interfacing with the seven-segment displays (see Figure 7) via CD4511s.The `lightDisplay()` function has two parameters both of which are 8-bit unsigned integers: digit, which can be from 1-4, and number, which is a single number from 0-9. The purpose of the function is to light the display specified by the digit parameter, with the number specified by the number parameter. For example, `lightDisplay(4, 8)` would show the number 8 on display number 4, which is the leftmost display. This is accomplished by storing the CD4511 pin assignments in a 2D array, and writing to them with 4 `digitalRead()` calls. Whether the bit is written high or low is determined by passing the "number" parameter through the built-in `bitRead()` function with the appropriate bit. For example, to write to bit 3 of digit 2 when the number 7 is passed to `lightDisplay()`, the following line would be used (note that arrays start at 0, not 1):

```
digitalWrite(CD4511_Pins[1][2], bitRead(7, 2));
```

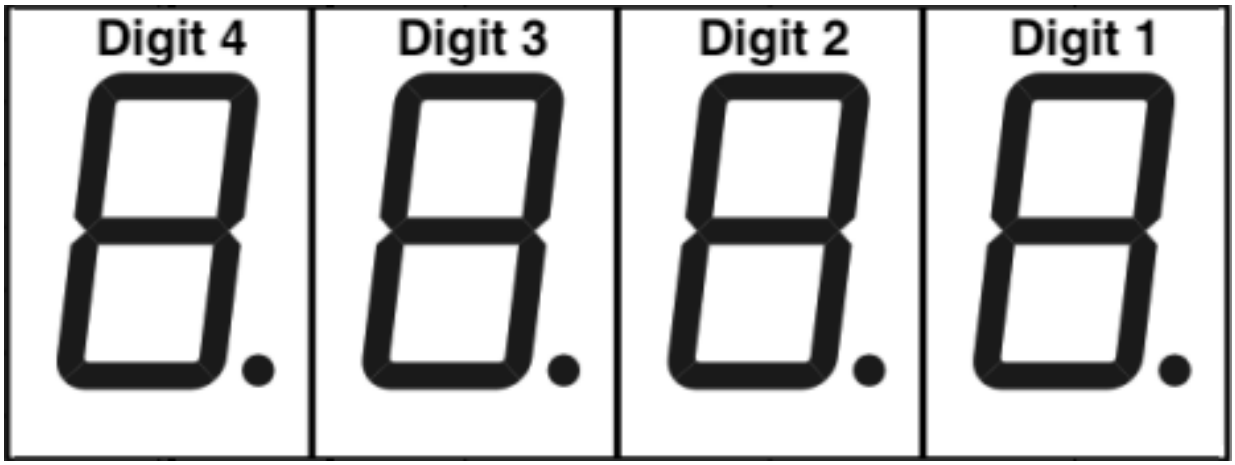



Figure 7. lightDisplay Digit Assignments

The second additional function, `readOut()`, had only one parameter: disp, a 16-bit unsigned integer. The purpose of the `readOut()` function is to display the value of disp (any number between 0 and 9999) on the set of four seven-segment displays (see Figure 7). It works by storing the given number into its 4 digits, each in a variable. It does this using the division and modulo operators. Then, it sets each digit to its respective variable using the `lightDisplay()` function (see Figure 8).

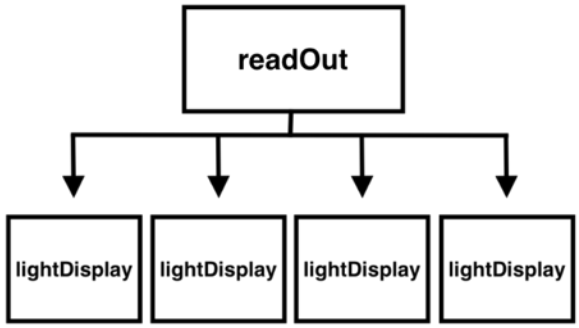



Figure 8. readOut to lightDisplay

The side of the bike computer's 3D printed case contains both a power switch, and a power barrel connector. The barrel connector is actually used as an info jack for the reed switch. The male connector is connected to the leads of the reed switch, while the connector has its wires going to the MCU (see Figure 9); this removably passes the reed switch to the MCU.


Figure 9. Info Jack

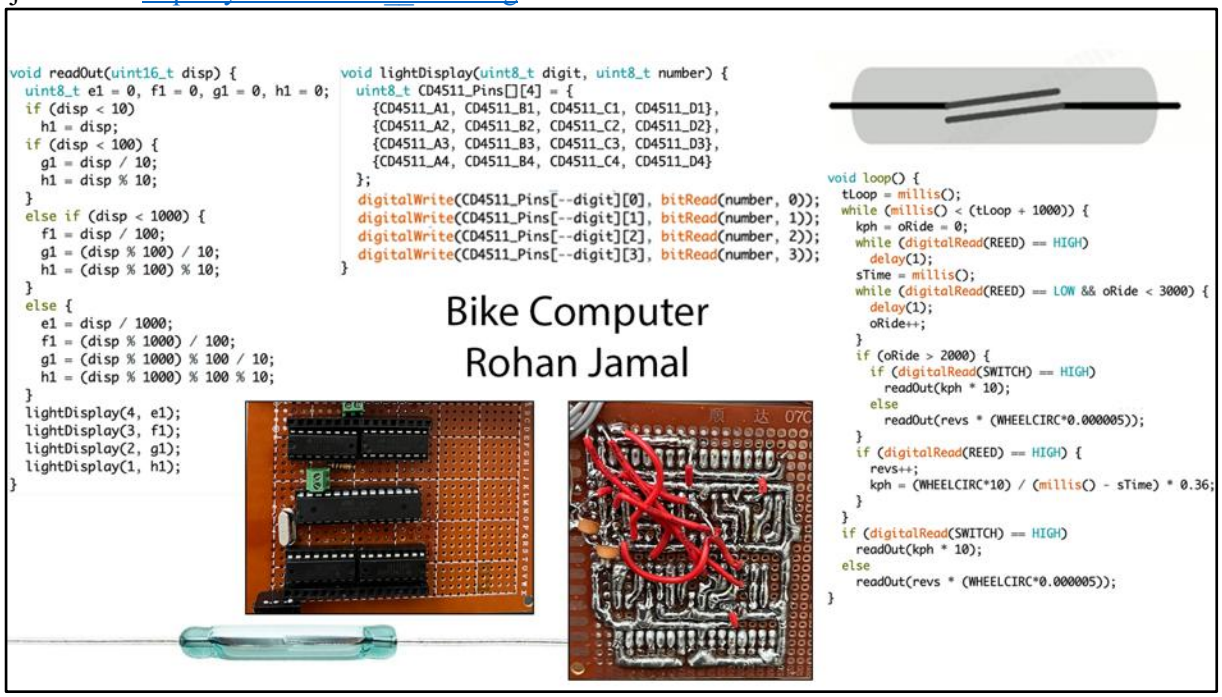
## Code

```
// PROJECT   : Bike speedometer
// AUTHOR    : R. Jamal
// PURPOSE   : To display the speed and distance of a bike on seven-segment displays
// COURSE    : ICS3U-E
// DATE      : 05 10 2023
// MCU       : 328P (DIP)
// STATUS    : Working
// REFERENCE : http://darcy.rsgc.on.ca/ACES/ISPs/Hardware.html
// NOTES     : Written in IDE 1.8.19. Uses reed switch for input to the bike
#define WHEELCIRC 2155                //Wheel circumference (in mm)
#define CD4511_A1   5
#define CD4511_B1   8
#define CD4511_C1   7
#define CD4511_D1   6
#define CD4511_A2   9
#define CD4511_B2   13
#define CD4511_C2   12
#define CD4511_D2   10
#define CD4511_A3   1
#define CD4511_B3   11
#define CD4511_C3   4
#define CD4511_D3   2
#define CD4511_A4   14
#define CD4511_B4   19
#define CD4511_C4   18
#define CD4511_D4   15
#define REED        17                //Pin for reed input from bike
#define SWITCH      16                //Pin for distance/speed switch
float kph;                            //Decimal-precision variable for KPH
uint64_t sTime, tLoop;                //Unsigned 64 bit integer: for millis()
uint32_t revs;                        //Unsigned 32 bit integer for revolutions
uint16_t oRide;                       //Unsigned 16 bit integer for override
void setup() {                        //Runs once, when powered on
  for (uint8_t i = 2; i <= 15; i++)   //Declare pins 2-15 as outputs
    pinMode(i, OUTPUT);
  pinMode(18, OUTPUT);                //Declare pin 18 as output
  pinMode(19, OUTPUT);                //Declare pin 19 as output
}
void loop() {                         //Runs on repeat after void setup()
  tLoop = millis();                   //Starts timer between readouts
  while (millis() < (tLoop + 1000)) { //Creates loop to enforce timer
    kph = oRide = 0;                  //Initializes necessary variables
    while (digitalRead(REED) == HIGH) //Wait for REED to go high then low
      delay(1);                       //Delays while REED is high
    sTime = millis();                 //Starts timer between revs
    while (digitalRead(REED) == LOW && oRide < 3000) {
      delay(1);                       //Wait 1 millisecond
      oRide++;                        //Count the ms the switch stays low
    }
    if (oRide > 2000) {               //Code if override has tripped
      if (digitalRead(SWITCH) == HIGH)
        readOut(kph * 10);            //Readout the stored speed, with decimal
      else                            //Reads out distance if in that mode
        readOut(revs * (WHEELCIRC*0.000005));     //Readout revs times circ
    }
    if (digitalRead(REED) == HIGH) {  //Executes when wheel has gone around
      revs++;
      kph = (WHEELCIRC*10) / (millis() - sTime) * 0.36;
    }
  }
  if (digitalRead(SWITCH) == HIGH)    //If in speed mode, readout speed
```

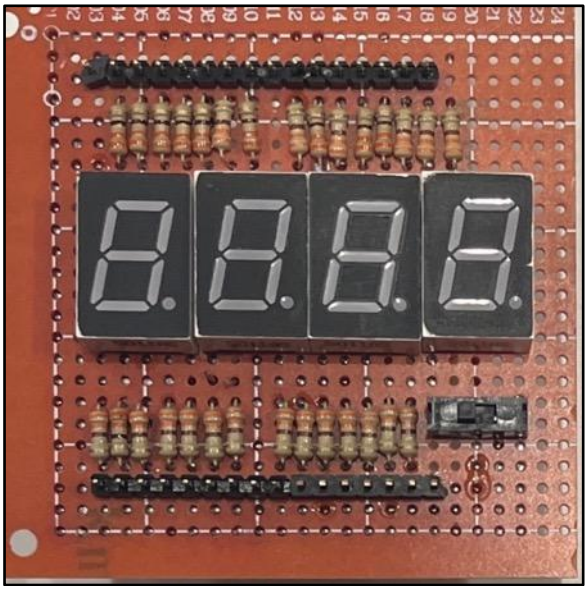
```
    readOut(kph * 10);                //Readout the stored speed, with decimal
  else                                //Reads out distance if in that mode
    readOut(revs * (WHEELCIRC*0.000005));   //Show revs times circ (distance)
}
void lightDisplay(uint8_t digit, uint8_t number) {
  uint8_t CD4511_Pins[][4] = {        //Defines 2D array for 4511 pins
    {CD4511_A1, CD4511_B1, CD4511_C1, CD4511_D1},
    {CD4511_A2, CD4511_B2, CD4511_C2, CD4511_D2},
    {CD4511_A3, CD4511_B3, CD4511_C3, CD4511_D3},
    {CD4511_A4, CD4511_B4, CD4511_C4, CD4511_D4}
  };
  digitalWrite(CD4511_Pins[digit - 1][0], bitRead(number, 0));  //LSB
  digitalWrite(CD4511_Pins[digit - 1][1], bitRead(number, 1));  //2nd LSB
  digitalWrite(CD4511_Pins[digit - 1][2], bitRead(number, 2));  //2nd MSB
  digitalWrite(CD4511_Pins[digit - 1][3], bitRead(number, 3));  //MSB
}
void readOut(uint16_t disp) {         //Reads out number in brackets
  uint8_t e1 = 0, f1 = 0, g1 = 0, h1 = 0;
  if (disp < 10)
    h1 = disp;                        //Gets number < 10 into h1
  if (disp < 100) {                   //Split number < 100 into digits
    g1 = disp / 10;                   //Separates high digit
    h1 = disp % 10;                   //Separates low digit
  }
  else if (disp < 1000) {             //Split number < 1000 into digits
    f1 = disp / 100;                  //Separates high digit
    g1 = (disp % 100) / 10;           //Separates middle digit
    h1 = (disp % 100) % 10;           //Separates low digit
  }
  else {                              //Split number > 1000 into digits
    e1 = disp / 1000;                 //Separates high digit
    f1 = (disp % 1000) / 100;         //Separates third digit
    g1 = (disp % 1000) % 100 / 10;    //Separates second digit
    h1 = (disp % 1000) % 100 % 10;    //Separates low digit
  }
  lightDisplay(4, e1);                //Lights up display with e1
  lightDisplay(3, f1);                //Lights up display with f1
  lightDisplay(2, g1);                //Lights up display with g1
  lightDisplay(1, h1);                //Lights up display with h1
}
```

## Media

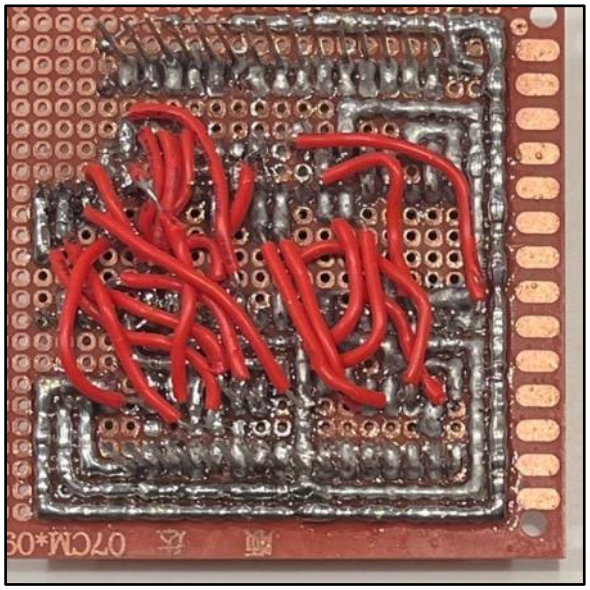
Project video: https://youtu.be/SSE__S8BEXg



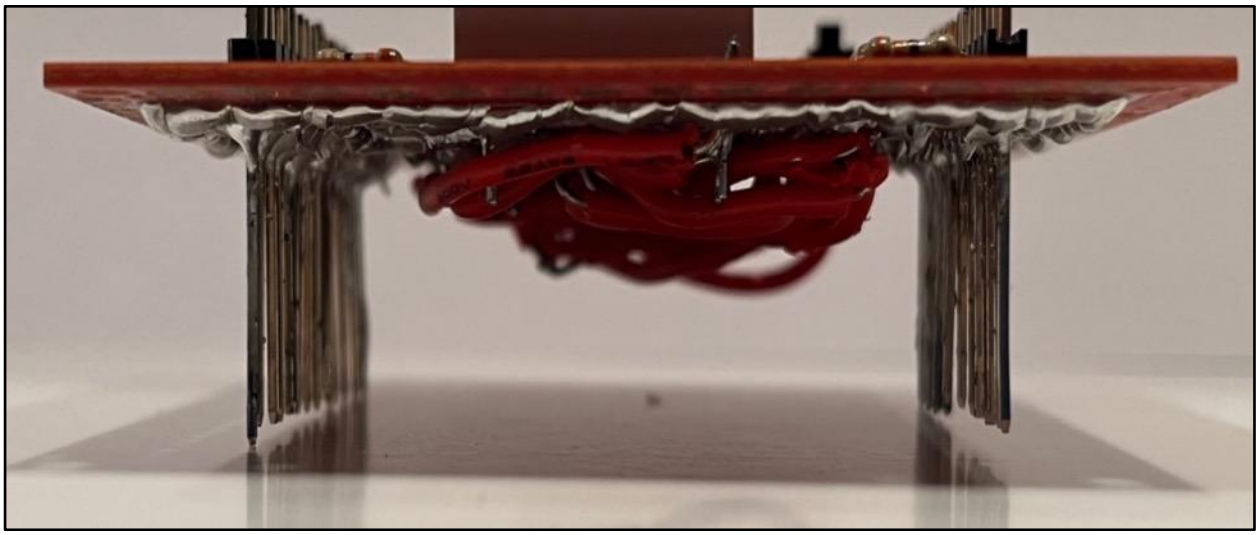

Presentation Background



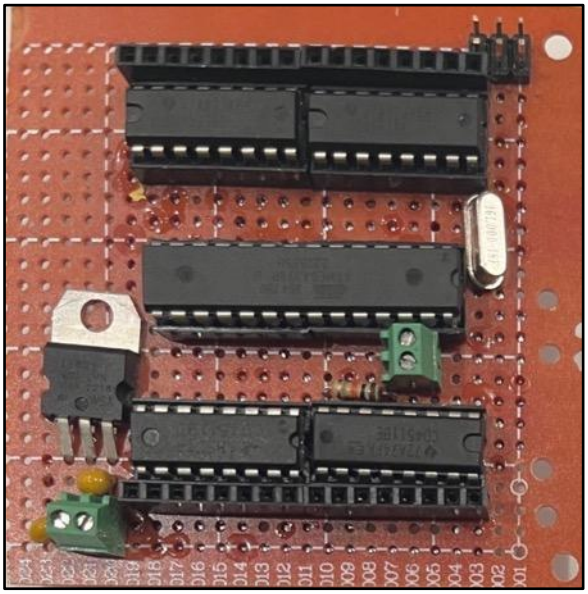
HID Board Top View



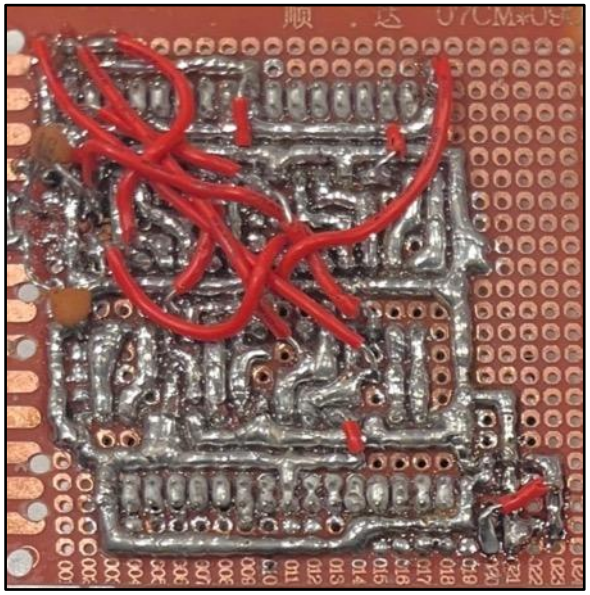
HID Board Bottom View



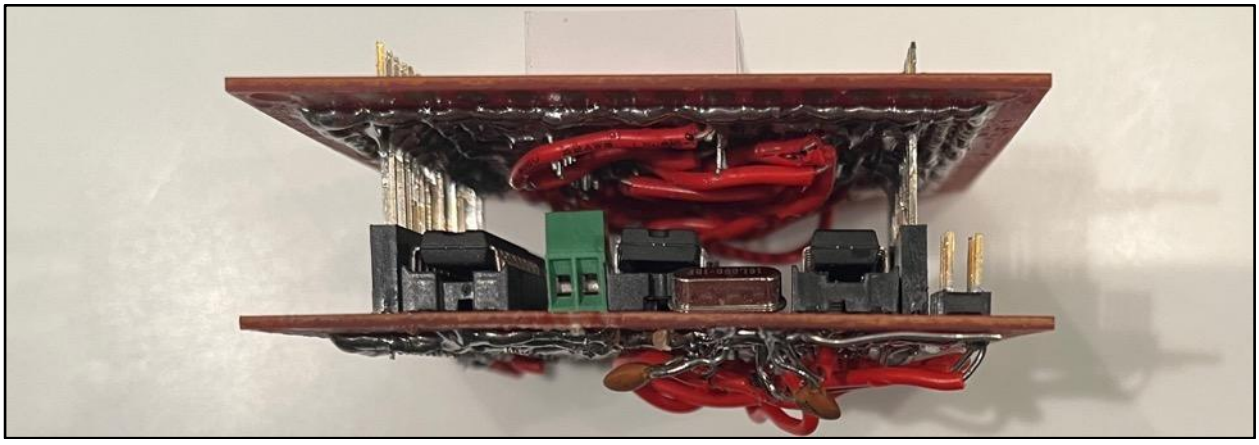
HID Board Side View



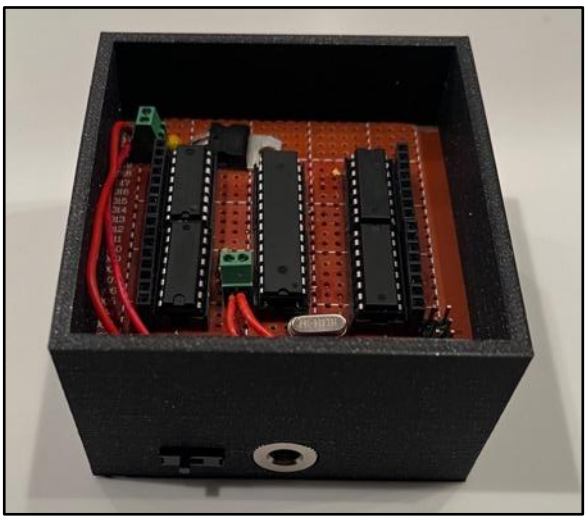
Internals Board Top View



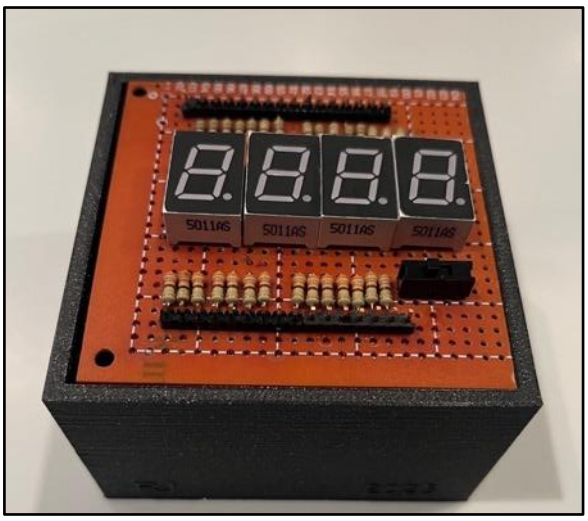
Internals Board Bottom View



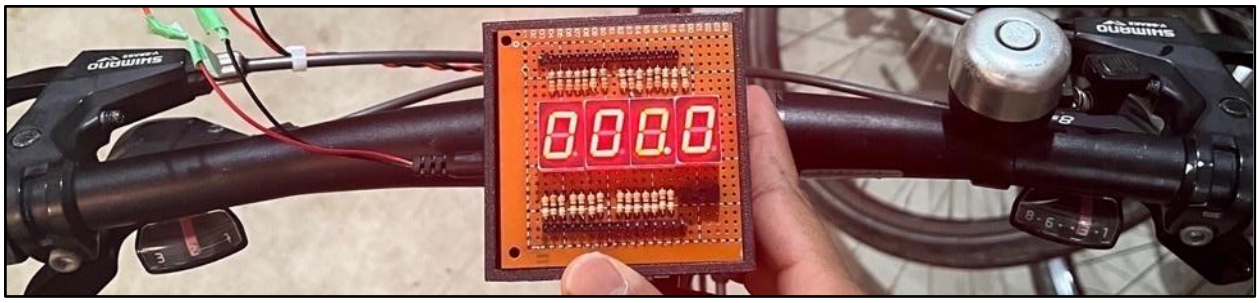
Entire Device Side Profile



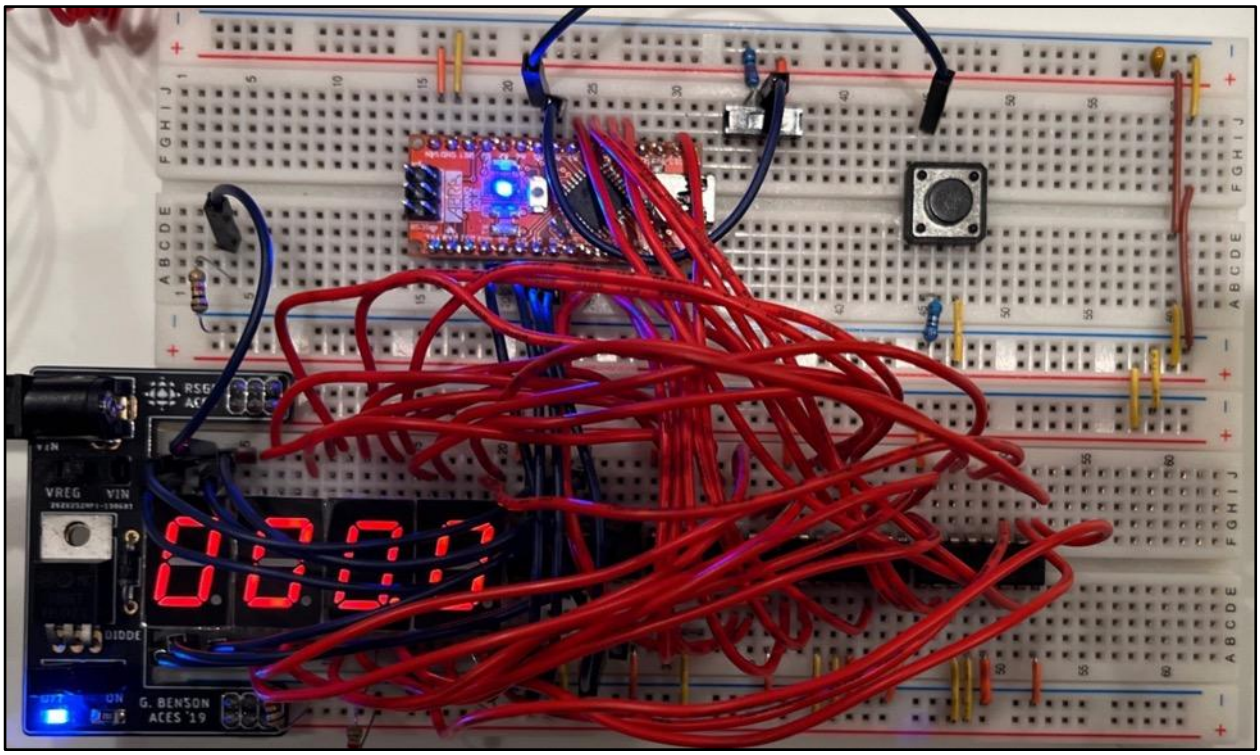
Inside the Case, Power Switch, Info Jack



Fully Assembled Device

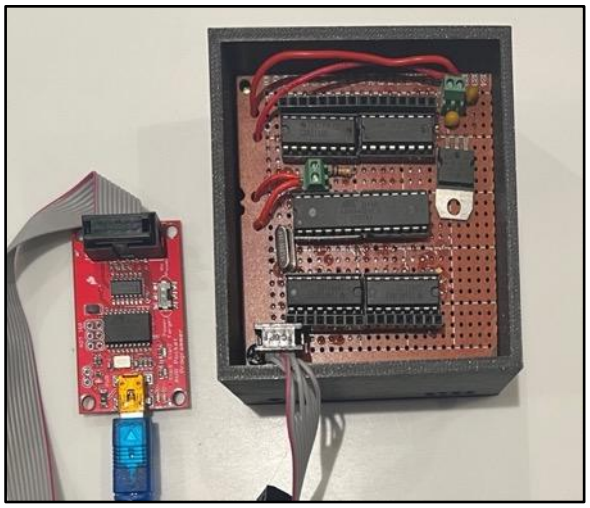


Device Connected to Info Jack on Bicycle



Original Prototype with PBNO in Place of Reed Switch and Arduino Nano

Reed Switch with Magnet



In-System Programmer (ISP) Connected to Header

## Reflection

This has definitely been my favourite project this year and last year. This is the first project where I have built something that has a practical function, and I guess it was just really refreshing to have a project in the background for a few months where I could just unplug from everything else and work on it. It's also very satisfying to have a finished project that I made from off-the-shelf components. I think that this project has launched me back into creating things. Before this, I was still creating things, like the required DERs, but it wasn't quite the same. I guess I think there's something to be said about having a project that isn't guaranteed to be possible. Starting with a hair-brained idea, turning it into a concept, and finally a fully working device is so much more interesting to me than simply following instructions to build a device. Don't get me wrong, there is value in both, I just think that every once in a while, it's important to go off the deep end and build something start to finish.

I genuinely think that this might be the first school project I have started more than a week before the due date, at least this year. If I'm being completely honest, I'm kind of surprised that my device even works. I knew that it would all work in theory, but there were just so many variables that could have gone wrong; like one of the million solder joints being compromised, or the last-minute addition of the voltage regulator, or jerry-rigging a power jack to transmit information. I guess an important lesson that I learned is to test each new addition; if you build everything, then power up the device at the end and something doesn't work, it's next to impossible to debug. In addition to starting well in advance, the key to this project's success was to test the circuit after each chip addition, display addition, board addition, or code update; when something went wrong, I knew exactly what part of the circuit to debug, and in most cases, could have the circuit up and running again within the hour.

As for things that did not work at all: about a month ago, when I got my first breadboard prototype working with a reed switch, I tried to change to a solid-state hall effect sensor for improved reliability. Unfortunately, I couldn't get it to work. No matter what I tried, the sensor wouldn't detect anything unless the magnet was almost touching it. I figured that since the reed switch was working, I could just leave it as is for the time being. It's been about a month with the same reed switch and magnet, and so far, it still works extremely accurately, almost never missing a pulse. Overall, this project has probably been the most successful yet, and I hope my next ISP (in the new year) can go as smoothly.

# Project 2.6 A Tiny Clock

## Project 2.6.1 Inter-Integrated Communication (I2C)

### Purpose

In addition to demonstrating the versatility of the Inter-Integrated Circuit (I2C) protocol, the purpose of the first part of the Tiny Clock (the I2C Bus) is to establish communication with both a Real Time Clock (RTC) and a temperature sensor over a single I2C bus.

### Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI3M/Tasks.html#TinyClock

Project schematic: https://crcit.net/c/3d373b7aa4c140348e7a6ef287c88e3a

### Theory

The I2C Bus is a communication protocol developed by Phillips Semiconductors (now NXP). Its purpose is to allow a master device to control up to 127 compatible devices using just 2 general-purpose input/output (GPIO) pins, referred to as serial data (SDA) and serial clock (SCL) (see Figure 1). The I2C protocol is able to control 127 devices by using 7-bit *addressing*. Addressing is the strategy used by I2C where each device is assigned a unique identifier called an address. This address is called upon every time the master calls on the device. In the case of 7-bit addressing, 128 addresses are theoretically available, however, address 0 is often reserved and thus cannot be used.

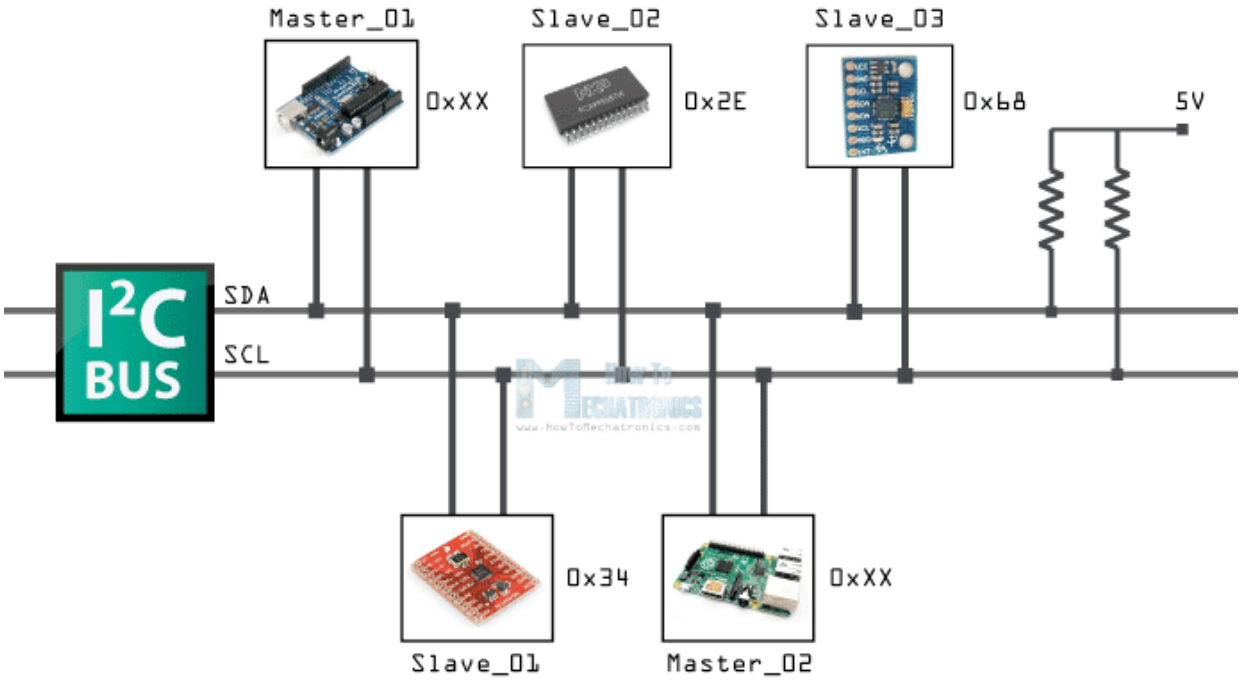



Figure 1. I2C Bus Lines

An RTC is an example of a device that can be made I2C compatible. This device is used to continue to keep time, even after the main power source of a system is terminated. Most RTCs utilize a $2^{15} = 32{,}768$ Hz crystal and have a small onboard coin cell battery to do so. This crystal is used for timekeeping as its frequency is a power of two. This means that the frequency can be easily and efficiently divided by 2 continuously to obtain a 1 Hz signal, allowing the RTC to update its time once every second. This optimal low crystal oscillation rate allows the RTC to draw an extremely low amount of current. As a result, RTCs are extremely versatile; they can keep time, often for years, without requiring a battery replacement.

## Procedure

The I2C Bus consists of an SCL and an SDA line (see Theory section). In this configuration, the SCL line oscillates at a constant frequency controlled by the master, 100 KHz by default. This line is used as the clock signal to synchronize the entire *network*. A network is a collection of multiple devices that can communicate with each other. Additionally, both the SCL and SDA lines are pulled high using pullup resistors. This is to overcome the fact that devices can only pull the SDA line low, not high. These resistors can be of any value, ranging from 2 KΩ on the low end, to 10 KΩ on the high end. On each high pulse of the SCL line, data is transmitted on the SDA line in 8-bit sequences (see Figure 1).

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | DS1307 RTC |
| 1 | 0.01 µF Ceramic Capacitor |
| 1 | ACES Coin Cell Breakout Board |
| 1 | 4 × 1 Male Headers |
| 1 | Metal Coin Cell Battery Bracket |
| 1 | TC74A5 Temperature Sensor |
| 2 | 10 KΩ Fixed Resistor |
| 1 | 32,768 Hz Crystal Oscillator |
| 1 | Arduino Nano |
| ~ | Solder |
| ~ | Wires |

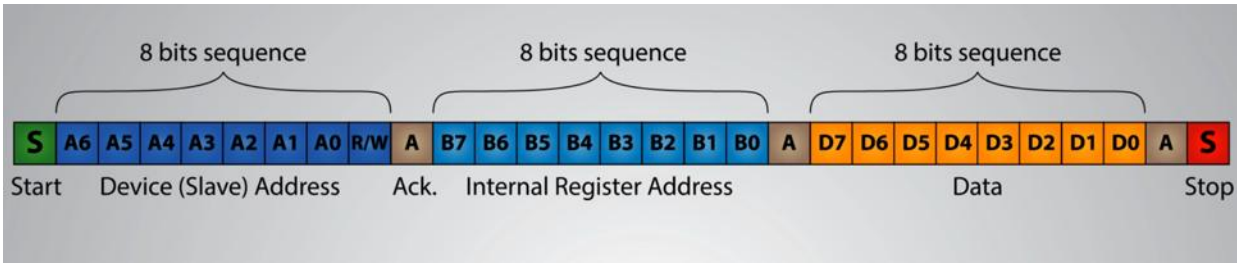



Figure 1. I2C Bus Bit Sequences

Figures 1 and 2 depict the order and timing in which bits are transferred between the slave and the master. First, the SDA line is pulled low before SCL transitions low. This is known as the start condition, and signals the devices to listen as a transmission is about to occur. After the start condition, the first 8-bit sequence is sent out. The first 7 bits are the 7-bit address of the desired device, and the final bit is the direction of data transmission. A high signal is indicative of a read operation, while a low signal pertains to a write operation. At the end of the byte, the master releases the SDA line, and waits for the receiver to pull the line low, to acknowledge the receiving of the byte (see Figure 2). The next byte pertains to the address of the register within the selected device. After this byte, another acknowledge sequence is performed. After this acknowledge, data is either read or written from the register, which is determined by the R/$\overline{W}$ bit (see Figure 2). This step is repeated as many times as necessary to read or write all the desired data, with an acknowledge sequence after every byte. When the transmission is complete, the master sends out the stop condition, signaled by both lines going idle.

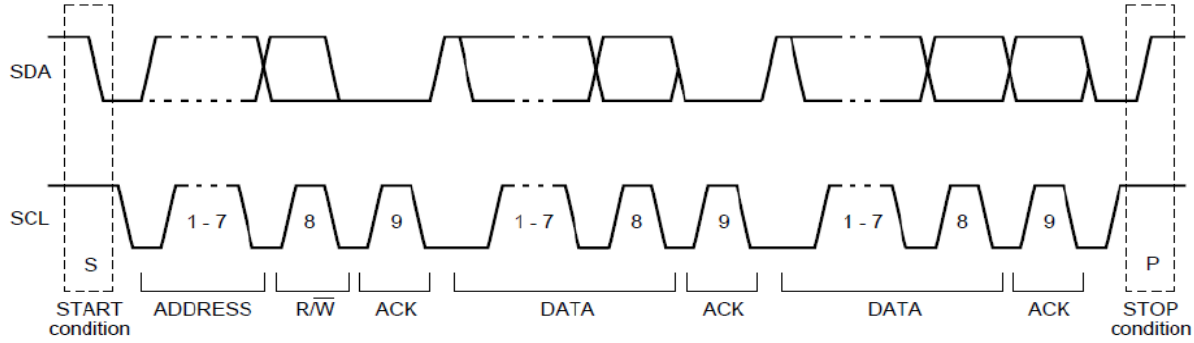



Figure 2. Activity on SDA and SCL Lines

As a result of each transmission requiring two bytes to setup, the speed depends on the length of the transmission. Multiple smaller-sized transmissions will result in more setup bytes per data byte, whereas a single long transmission will result in 2 setup bytes, and many data bytes (see Figure 3). The more bytes of data that are transmitted, the closer the transmission is to being 100% data bytes.

| Figure 3. Transmission Sizes | |
|---|---|
| Data Bytes | Total Bytes (No Ack) |
| 1 | 3 |
| 10 | 12 |
| 50 | 52 |
| 100 | 102 |
| 500 | 502 |

An I2C network can be conceptualized as a 3-dimensional network as there are three distinct layers to it; addresses, registers, and data (see Figure 4). The first layer, addresses, refers to the highest-level organization of the network, the physical device. Within each address, or physical device, there are multiple registers, which are the second dimension of an I2C network. Each register is another location within the physical device. The third dimension is the actual data being read or written, which is one level lower than the register. The data is analogous to the register in the same way that the register is analogous to the address; the data is a location within the register just as a register is a distinct location within a specified address (see Figure 4).

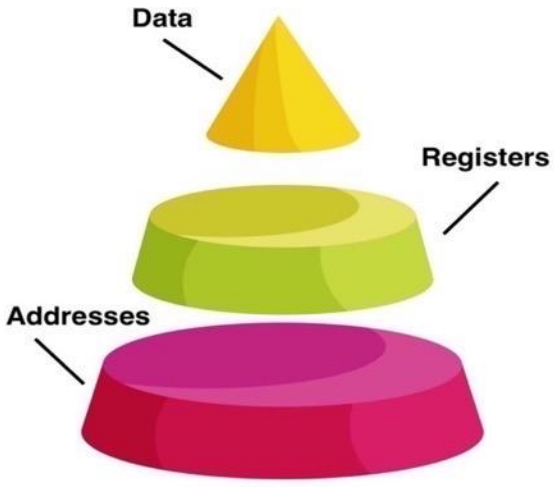



Figure 4. I2C Dimensional Visualization

On the Arduino Nano, pin A4 is SDA and pin A5 is SCL (see Figure 5). The code written to scan an I2C bus for devices on the Nano (see Code section) utilizes the Wire library. Using behind-the-scenes, prewritten code, the Wire library allows the user to easily utilize the I2C protocol with calls to various functions.

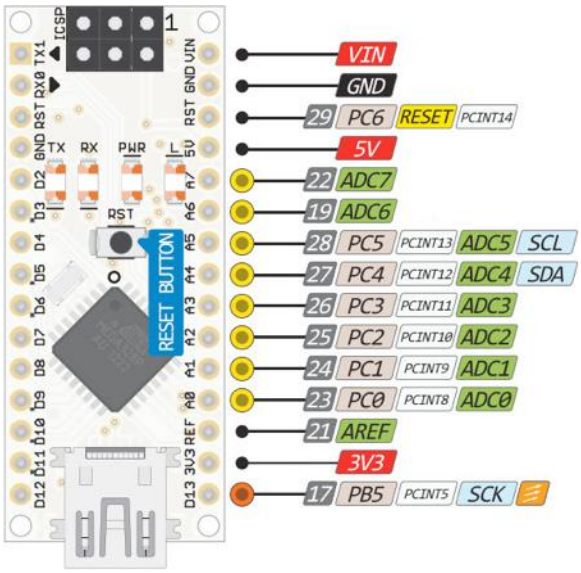



Figure 5. Arduino Nano Pinout

Within the setup function of the code, the `begin()` function is used to initialize the Wire library. Next, the code simply employs a `for()` loop to run through addresses 0x01 to 0x7F (1-127), passing the address to the `beginTransmission()` function. The program immediately passes the same address to the `endTransmission()` function, declaring a device as found if the function returns a 0, as outlined in the documentation

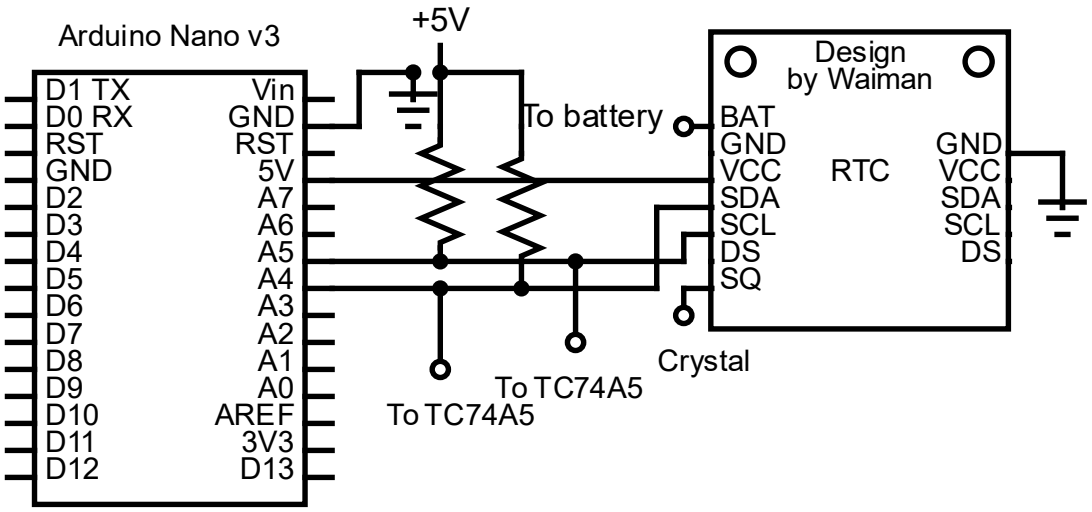



Figure 6. I2C Scanner Arduino Schematic

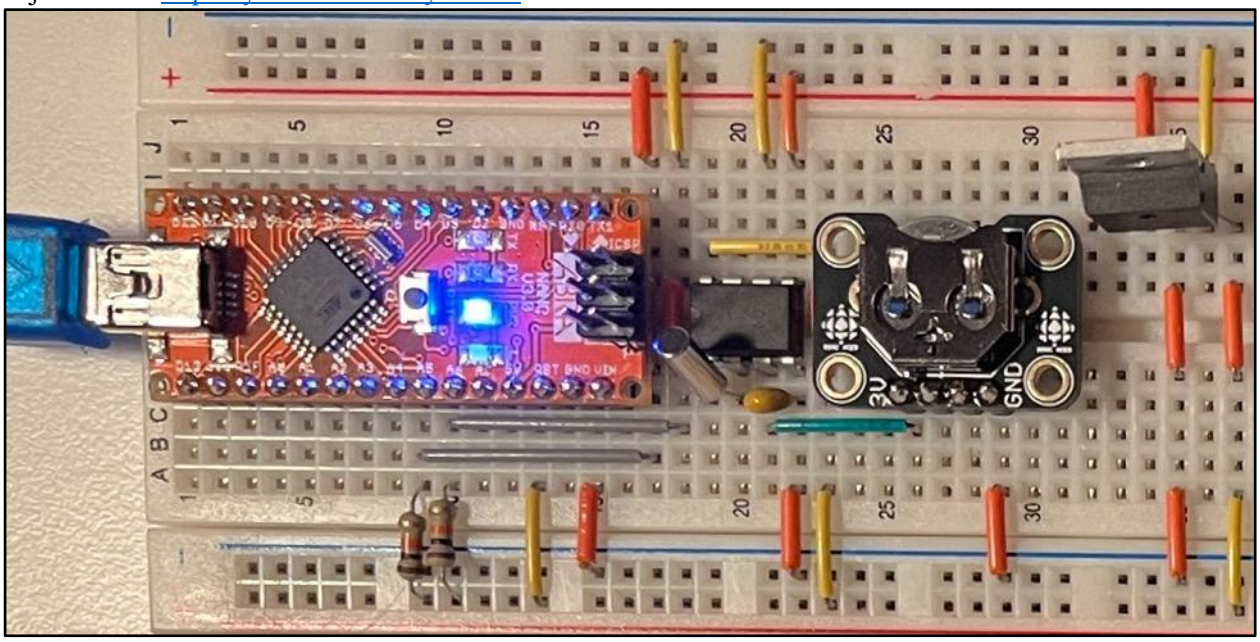
Figure 6 depicts how the I2C scanner is constructed using an Arduino Nano. In this configuration, the RTC is connected to both the power from the Arduino and the battery backup. The SCL (A5) and SDA (A4) lines are connected to the appropriate pins on the RTC, and most notably, the same lines are both pulled high, and connected to the TC74A5 temperature sensor.

## Code

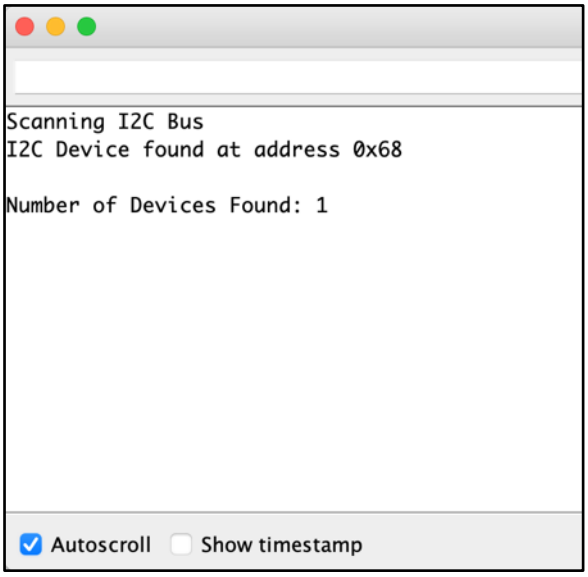
```
// PROJECT   : I2CScanner
// AUTHOR    : R. Jamal
// PURPOSE   : To scan an I2C bus for devices
// COURSE    : ICS3U-E
// DATE      : 2024 01 25
// MCU       : 328P (Nano)
// STATUS    : Working
// REFERENCE : http://darcy.rsgc.on.ca/ACES/TEI3M/Tasks.html#TinyClock
// NOTES     : Uses wire library, written in Arduino IDE 1.8.19
#include <Wire.h>       //Handles sync signals
#define INTERVAL 5000   //Define how often to scan
uint8_t numDevices;     //Variable to store how many devices found
void setup() {
  Wire.begin();         //Initializes I2C bus on A4 and A5
  Serial.begin(9600);   //Initializes Serial communication
  while (!Serial);      //Waits for Serial Monitor to open
}
void loop() {
  numDevices = 0;                         //Sets devices found between scans
  Serial.println("\nScanning I2C Bus");   //Prints in Serial
  for (uint8_t address = 1; address < 128; address++) {
    Wire.beginTransmission(address);      //Starts transmission to "address"
    if (!Wire.endTransmission(address)) { //Stops transmission
      numDevices++;                       //Adds a device to total devices found
      Serial.print("I2C Device found at address 0x");   //Prints in Serial
      if (address <= 0x0F)                //Print 0 if address is less than 16
        Serial.print('0');                //Leading hex 0 for addresses less than 16
      Serial.println(address, HEX);       //Prints the address of the device in hex
    }
  }
  Serial.println("\nNumber of Devices Found: "+String(numDevices)); //# device found
  delay(INTERVAL);                        //Kills clock cycles for INTERVAL
}
```

## Media

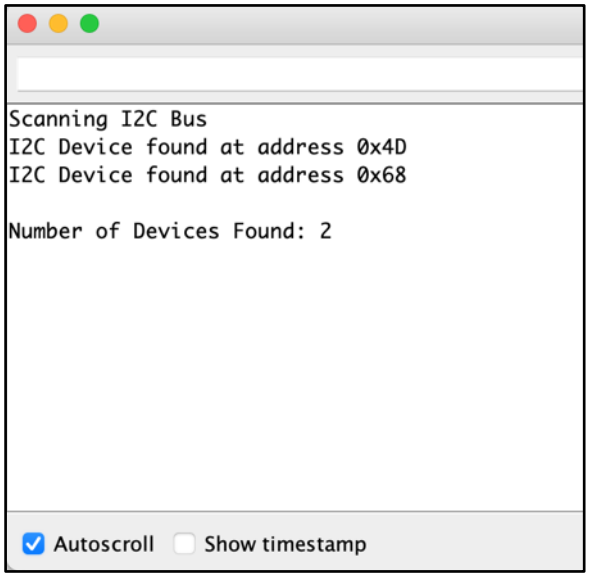
Project video: https://youtu.be/ZriuUymdlO0



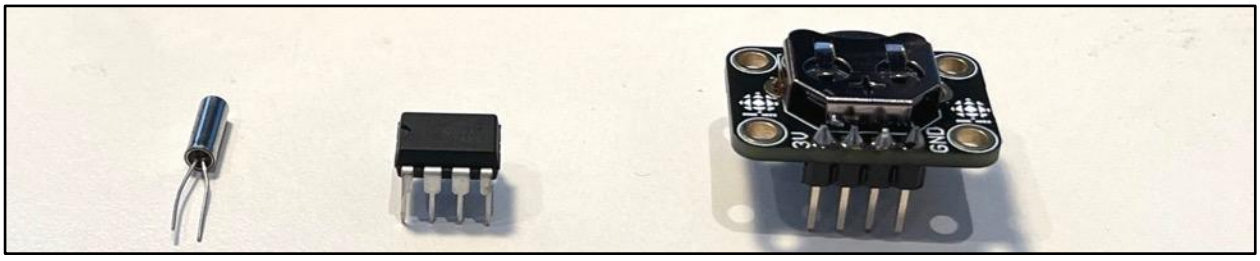

Scanner Circuit Overview




Serial Monitor RTC Connected




Serial Monitor RTC and TC74A5 Connected




RTC (Middle) and Supporting Components Crystal (Left) and Battery Breakout Board (Right)

## Reflection

When we started this project, I thought it was pretty cool. I think this is the first "high-level" project we've done. By that I mean the first software project that was something that serves a practical purpose in everyday life. So naturally, when Mr. D'Arcy expressed his dislike for the Wire library, I attempted to rectify the situation.

It was the Friday before the project was due, and I had had a late night the night before. I was in English class, and could barely stay awake. My thoughts somehow drifted to the I2C bus, and a video we had watched explaining the bit transfer sequence of I2C, and the fact that there was a clock signal (square wave) to synchronize the bus. That was when a sudden wave of energy and clarity hit me. I thought of the blink sketch, and how it was really just a square wave, but more importantly, that we had just rewritten the sketch using interrupts. This meant (or at least I thought) that I could have a blink sketch running on the SCL pin, while doing something entirely different on another pin (SDA, for example). Using the knowledge of the I2C bit sequence in tandem with this, I reasoned, could allow me to write an I2C scanner without the Wire library. After thinking about it for a few minutes, I had (what I though was) a solid plan in my head: step 1: start the blink sketch on SCL in the setup function. Step 2: send out the 7-bit address on SDA, followed by a high to put it in read mode. Step 3: set the SDA to idle, and if the line was pulled low, there was indeed a device on that address. If not, it would check the next address.

When I got home, I immediately began writing the code, based off of the sketch with the Wire library that I already had. After about an hour, I had my code ready to test. That was when the first issue arose: the interrupt strategy would cause the clock signal to stop every time something else was supposed to happen. Unfazed, I replaced the SCL line with pin 3, and used the `tone()` function to generate the 100 KHz clock signal instead. This was when my next problem came up: I didn't have a solid way to make sure SDA was synchronized with the clock signal. I decided to tie pin 3 to an analog input, and used a while statement to wait for the high pulse. I'm not sure if that's what ultimately rendered my code unusable, but I never got it working. My code was plagued by issues. I figured if I really wanted, I could have done it manually with a call to `digitalWrite()` every 5 microseconds, using delays and setting SDA with the same method; I decided, however, that it didn't make sense and would just be inefficient, so I stuck with the regular Wire code. Even though my code seemed foolproof to me, most versions didn't find any devices, for some reason. Only one version of the code found devices, but it found the same 54 addresses, every time. Of course, most of these were incorrect. I figured it must have been a result of SDA and SCL not being synchronized properly.

Overall, this project has been somewhat bittersweet; it introduced I2C, and I fully understand how it works on a technical level, but at the same time I wish I could have gotten a scanner working with my custom code. I guess maybe it's for the best. It taught me that I still have lots to learn, and the good news is, I'm confident that if I keep developing my software skills, one day, I'll look back at this experience, and easily be able to write up the scanner code.

## Project 2.6.2 Real Time Clock (RTC) Prototype

### Purpose

In addition to keeping the passage of time without external power, the purpose of the RTC Prototype is to display the current time and date on a 4 digit seven-segment display using an ATtiny85 and two shift registers.

### Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI3M/Tasks.html#Prototype

### Theory

Persistence of vision (POV) refers to the effect of visual perception of an object lingering even after light rays from the object stop being emitted. For example, when an object is moved back and forth quickly, multiple versions of the object are perceived. This means that an interrupted light source, when interrupted frequently, will appear to remain constantly present. This means that two light sources can be displayed in rapid succession and will appear to both be present 100 percent of the time. Additionally, the duty cycle, or percentage of time on to total time, can be used to control the perceived brightness. As a result of human eyes being more sensitive to light changes in dark environments than bright ones, a 50 percent duty cycle does not correlate with a 50 percent perceived brightness. As the duty cycle is steadily decreased from 100 percent to approximately 30 percent, the percentage at which the perceived brightness diminishes is nearly directly proportional, with a 30 percent duty cycle correlating to a 55 percent perceived brightness (see Figure 1). The duty cycle ranging from 0-5 percent correlates with a perceived brightness control between 0 and 20 percent.

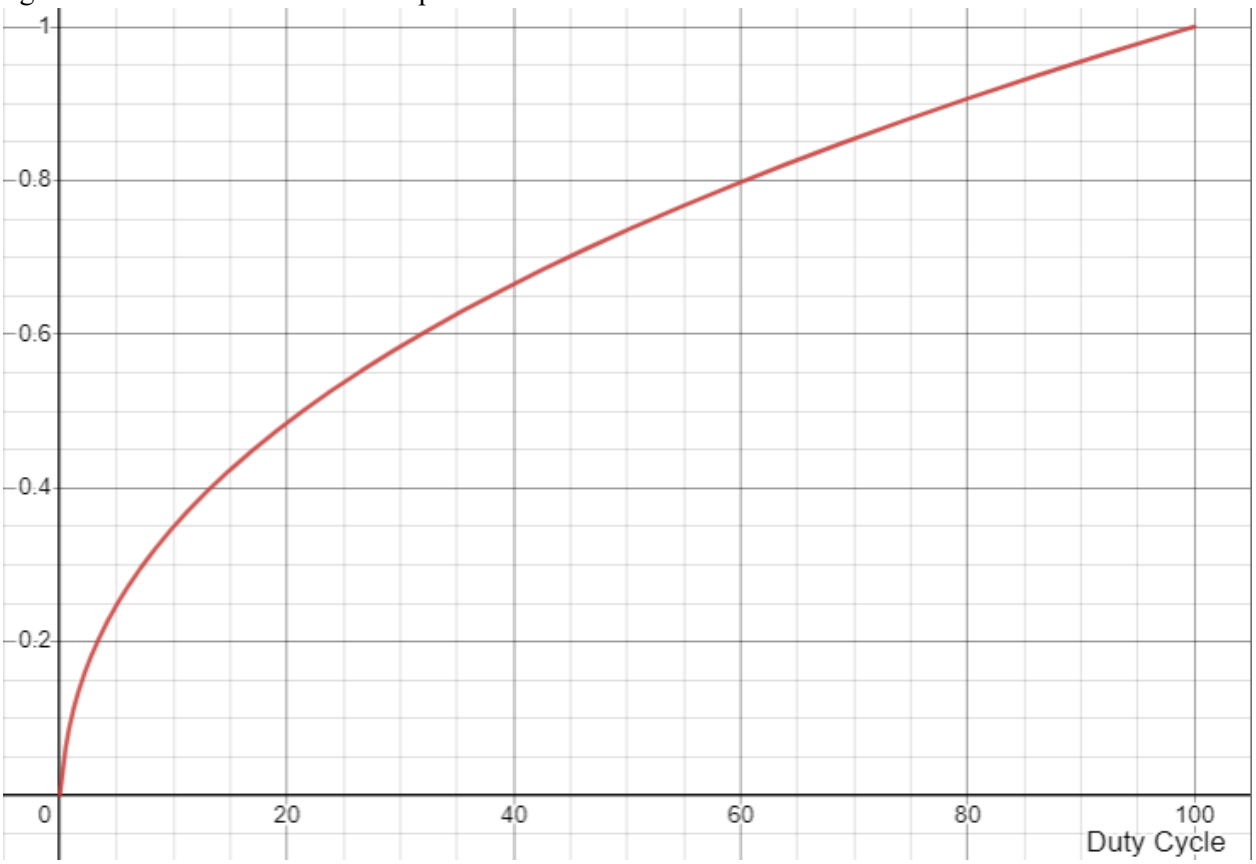



Figure 1. Duty cycle vs Perceived Brightness

## Procedure

The RTC Prototype displays the current time on a common anode 4 digit seven-segment display, and can continue to internally keep the passage of time when power is disconnected. It does this using a real-time clock (RTC). An RTC is a dedicated chip with a backup coin cell battery that keeps track of time based on oscillations of a known frequency from a crystal (see Project 2.6.1). The current time is stored in a series of registers accessible over the I2C protocol (see Project 2.6.1).

To convert the time stored in the registers of the RTC to a human-readable time on the display, an ATtiny85 8-pin MCU is utilized. The MCU reads the time from the RTC, then displays the received value on the display. The display requires 11 pins to drive the digits, so two shift registers are utilized to extend I/O (see Figure 1). The first controls the cathodes, while the second controls the anodes.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 5 | 2N3906 PNP Transistor |
| 5 | 10 KΩ Fixed Resistor |
| 1 | 5.1 KΩ Fixed Resistor |
| 2 | 2.2 KΩ Fixed Resistor |
| 9 | 470 Ω Fixed Resistor |
| 1 | Light-Dependant Resistor |
| 1 | ATtiny85 DIP MCU |
| 2 | SN74HC595N Shift Register |
| 1 | 4 Digit 7-Segment Display |
| 1 | RSGC ACES RTC BoB |
| 1 | DS1307 RTC |
| 1 | 0.1 µF Ceramic Capacitor |
| 1 | Signal Diode |
| 1 | Coin Cell Battery |
| 1 | 32,768 Hz Crystal Oscillator |
| ~ | Wires |

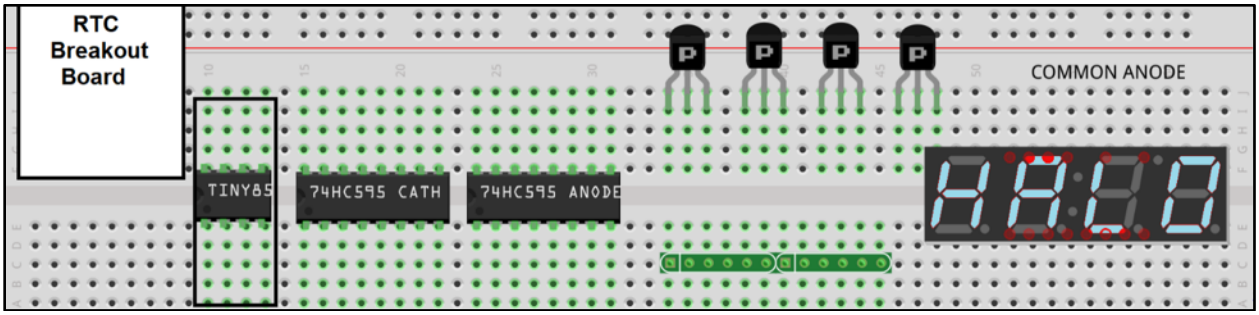


Figure 1. RTC Breadboard Layout

When displaying a number, the MCU shifts out the anodes first. Each of the high 4 bits corresponds to a distinct digit within the display. A digit is activated by grounding a bit due to the use of PNP transistors. In this configuration, the segments required to light each digit are stored in an array, from 0-9, with the 10th being all segments off (see Figure 2). Each bit corresponds to a high pin on the shift register, or a segment lit. The LSB is segment g, while segment a is bit 7. The MSB of the cathode bit is unused. The cathode byte must be inverted before being shifted as it activates a segment with a path to ground. Each digit is shifted out rapidly, creating the effect that all 4 are constantly lit up (POV) (see Theory section).

| Figure 2. Segment Mappings | | |
|---|---|---|
| **Number** | **Segments** | **Value (BIN)** |
| 0 | a, b, c, d, e, f | 01111110 |
| 1 | b, c | 00110000 |
| 2 | a, b, d, e, g | 01101101 |
| 3 | a, b, c, d, g | 01111001 |
| 4 | b, c, f, g | 00110011 |
| 5 | a, c, d, f, g | 01011011 |
| 6 | a, c, d, e, f, g | 01011111 |
| 7 | a, b, c | 01110000 |
| 8 | a, b, c, d, e, f, g | 01111111 |
| 9 | a, b, c, d, f, g | 01111011 |
| OFF | None | 00000000 |

Before the time can be read, the RTC must be programmed. The RTC is programmed by storing the binary-coded decimal (BCD) number in the corresponding register.

BCD is a specifically encoded version of binary, where each nibble corresponds to a decimal digit (see Figure 3). For example, the number 12 would be 0001 0010. In this configuration, the ones digit, 2, is represented by the low nibble. The tens digit, 1, is represented by the high nibble.

To convert the desired number from decimal to BCD, the function `BCD2DEC(uint8_t value)` is used, accepting 1 parameter: value. The function simply shifts the tens digit left 4 times, or multiplies by 16, and adds it to the ones digit. This converts the number to BCD by storing the tens digit in the high nibble, and the ones digit in the low nibble.

**Figure 3. BCD Truth Table**

| Decimal | BCD |
|---|---|
| 0 | 0000 0000 |
| 1 | 0000 0001 |
| 2 | 0000 0010 |
| 3 | 0000 0011 |
| 4 | 0000 0100 |
| 5 | 0000 0101 |
| 6 | 0000 0110 |
| 7 | 0000 0111 |
| 8 | 0000 1000 |
| 9 | 0000 1001 |
| 10 | 0001 0000 |
| 11 | 0001 0001 |
| 12 | 0001 0010 |
| 13 | 0001 0011 |
| 14 | 0001 0100 |
| 15 | 0001 0101 |

| ADDRESS | BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 | FUNCTION | RANGE |
|---|---|---|---|---|---|---|---|---|---|---|
| 00h | CH | 10 Seconds | | | Seconds | | | | Seconds | 00–59 |
| 01h | 0 | 10 Minutes | | | Minutes | | | | Minutes | 00–59 |
| 02h | 0 | 12 | 10 Hour | 10 Hour | Hours | | | | Hours | 1–12 +AM/PM 00–23 |
| | | 24 | PM/ AM | | | | | | | |
| 03h | 0 | 0 | 0 | 0 | 0 | DAY | | | Day | 01–07 |
| 04h | 0 | 0 | 10 Date | | Date | | | | Date | 01–31 |
| 05h | 0 | 0 | 0 | 10 Month | Month | | | | Month | 01–12 |
| 06h | 10 Year | | | | Year | | | | Year | 00–99 |
| 07h | OUT | 0 | 0 | SQWE | 0 | 0 | RS1 | RS0 | Control | — |
| 08h–3Fh | | | | | | | | | RAM 56 x 8 | 00h–FFh |

Figure 4. DS1307 Registers

The RTC contains eight programmable useful registers (see Figure 4). The first seven pertain to time-related attributes, while the eight is used to set a square wave on pin 7 (see Figure 5). To program the time and date, each value is sent to the corresponding register over I2C. Once the time has been set, the RTC will keep the passage of time.

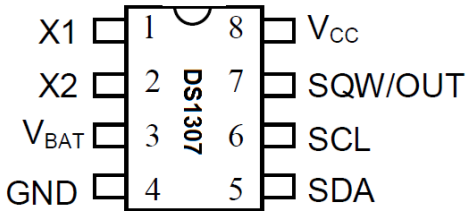



Figure 5. DS1307 Pinout

The square wave register (register 7) has three important bits to be set: bits 4, 1, and 0 (see Figure 4). If bit 4 is high, a square wave is presented on pin 7 (see Figure 5). When bit 4 is low, pin 7's logic level can be set with bit 7. Additionally, bits 1 and 0 (formatted RS1, RS0) control the frequency of the square wave from four possibilities: 1 Hz (0, 0), 4096 Hz (0, 1), 8192 Hz (1, 0), and 32,768 Hz (1, 1).

There are two main methods of setting the time on the RTC. The first is manually, with hard-coded values that must constantly be updated. The second programming method is with date and time *macros*. A macro is a large chunk of code that can become available to the programmer from a relatively small sequence of characters.

The `sscanf()` function can be used to parse the time and date macros into useable data. For example: `sscanf(__TIME__, "%d:%d:%d", &hrs, &mins, &secs)` parses the data returned by the time macro (`__TIME__`). The format that the macro returns is specified in the second parameter, and the variables in which to store the parsed data are given in the third parameter. This statement effectively stores the current hours to hrs, minutes to mins, and seconds to secs. Similarly, the `__DATE__` macro can be used to parse the day month, date, and year into variables to be sent to the RTC over I2C.

The RTC Prototype is configured to present a 1 Hz square wave. This square wave is pulled up, and fed through a fixed 470 Ω resistor into cathode of the colon LED (see Figure 6). This causes it to flash at a rate of 1 Hz. Additionally, the anode of the colon LED is connected to a PNP transistor for brightness control. In this configuration, the base pin of the transistor is tied to the same PWM pin on the ATtiny85 as the OE pin on the shift registers. This allows the brightness of the digits to match that of the colon.

In addition to the square wave pin, the cathode is also connected to anode shift register output 1 through a signal diode, allowing it to be switched off without affecting the digits. This allows the colon to be switched off when displaying the date.

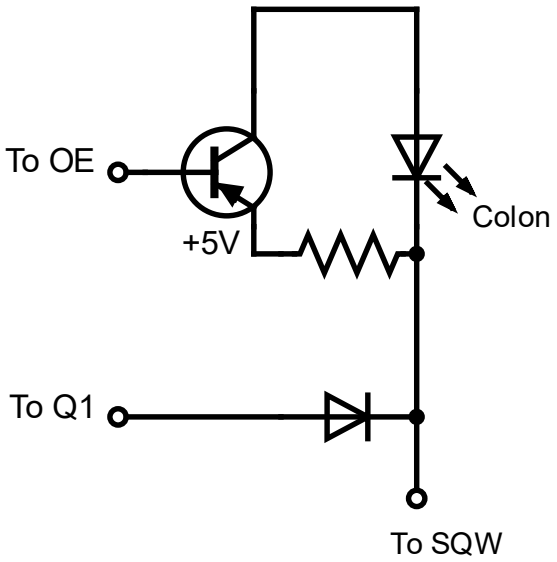


Figure 6. Colon Schematic

When displaying the date, the ATtiny85 switches off the colon, and replaces it with the decimal point. In this configuration, the date is displayed in MM.DD format. For example, on December 1, it would display 12.01. This prevents inconsistencies in lettering on seven-segment displays, as not all letters can be properly displayed and is accomplished by connecting the decimal point cathode to cathode shift register output 1 (see Figure 7).

When provided a path to ground, the decimal point lights up. Additionally, since the decimal point is connected to the shift registers, unlike the colon, no transistor is required to maintain brightness control.

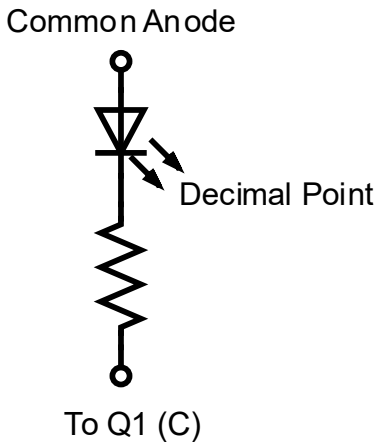


Figure 7. Decimal Point Schematic

Brightness on the RTC prototype is controlled using PWM. In this configuration, a PWM-capable pin is connected to both shift register output enable (OE) pins, as well as the base pin of the PNP transistor. By adjusting the duty cycle of the PWM square wave (see Figure 8), the persistence of vision effect causes the perceived brightness of the display to change (see Theory section). The written brightness is determined by a light-dependant resistor (LDR) in a voltage divider configuration. An analog pin reads the voltage on the voltage divider, which changes according to ambient light. The duty cycle of the LEDs changes appropriately in order to remain the correct brightness for the amount of ambient light present.

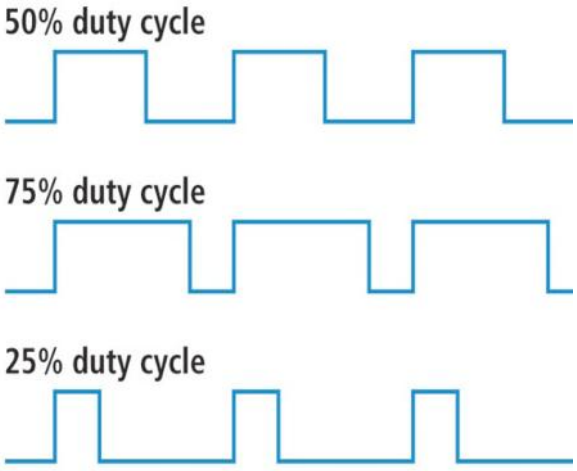



Figure 8. Duty Cycles

Due to the limited number of I/O pins on the ATtiny85, the shift register clock pin, which is on port B, is used to read the voltage of the LDR (see Figure 9). In this configuration, the data direction register for port B (DDRB) must be updated every time the sensor is read. When reading the brightness, the clock pin bit on DDRB is set to input (low). When writing to the display, it is set to output (high). To set a bit, the OR operator is utilized. To clear a bit, the AND operator is used with both a mask and the NOT operator.

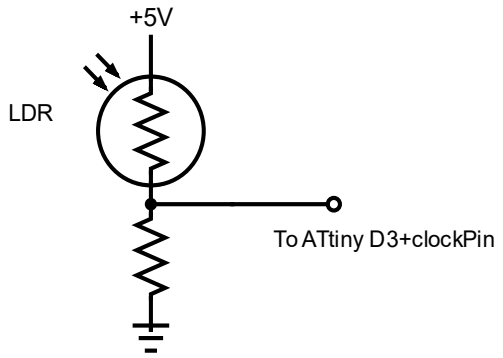



Figure 9. Light-Dependant Resistor Schematic

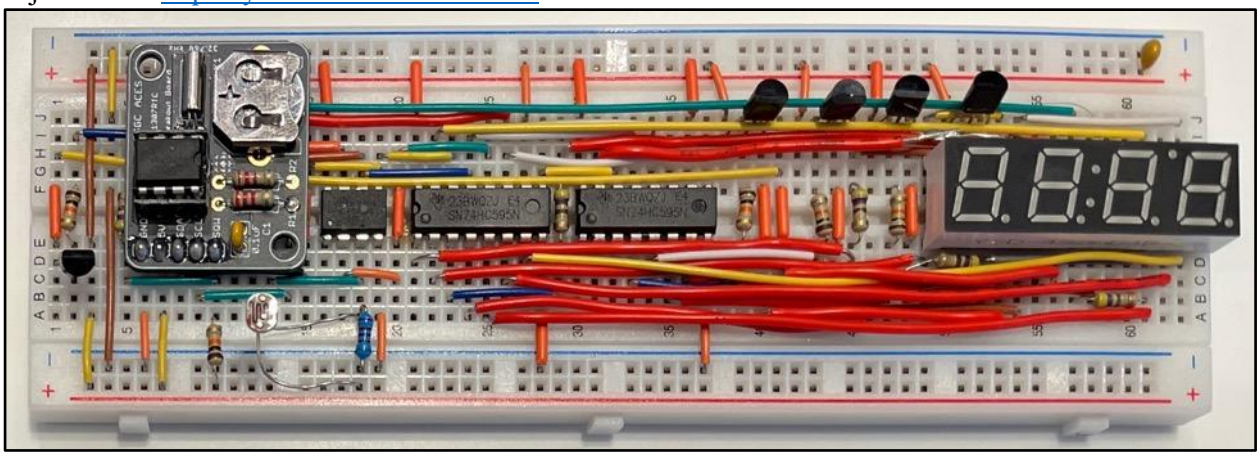
## Code

```
// PROJECT   : Tiny Clock Breadboard
// PURPOSE   : To display the time/date using Attiny85 on a seven segment display
// COURSE    : ICS3U
// AUTHOR    : R. Jamal
// DATE      : 2024 02 17
// MCU       : '85
// STATUS    : Working
// REFERENCE: http://darcy.rsgc.on.ca/ACES/TEI3M/Tasks.html#Prototype
// Notes     : Uses millis() to show date. Interrupts unavailable; PWM uses timer1
#include <TinyWireM.h>              //Include the TinyWireM lib
#define RTCADDRESS  0x68            //I2C address of RTC
#define clockPin 4                  //Clock pin shift register
#define latchPin 3                  //Latch pin shift register
#define dataPin 0                   //Data pin shift register
#define BRIGHT A2                   //Photoresistor pin (D4=A2)
#define OE 1                        //Output enable pin (brightness control)
#define BOV 215                     //Blackout value (brightness in pitch-black)
#define COLON 7                     //Colon connection on shift register (bit 8)
uint8_t date = 0;                   //Keeps track of which stat to display
uint64_t timeBuffer;                //Used to keep track of time with millis()
uint8_t buffer[] = {4, 3, 2, 1};    //Intermediate data holder
uint8_t decbit;                     //Variable to hold decimal state
uint8_t numbers[] = {               //Start segment map
```

```
  B01111110,                        //0 (a,b,c,d,e,f)
  B00110000,                        //1 (b,c)
  B01101101,                        //2 (a,b,d,e,g)
  B01111001,                        //3 (a,b,c,d,g)
  B00110011,                        //4 (b,c,f,g)
  B01011011,                        //5 (a,c,d,f,g)
  B01011111,                        //6 (a,c,d,e,f,g)
  B01110000,                        //7 (a,b,c)
  B01111111,                        //8 (a,b,c,d,e,f,g)
  B01111011,                        //9 (a,b,c,d,f,g)
  B00000000                         //Off (No segments)
};                                  //End segment map
void setup() {
  TinyWireM.begin();                //Initialize I2C
  DDRB |= 1 << latchPin | 1 << clockPin |
          1 << dataPin  | 1 << PB2; //Set I/O
}
void loop() {
  timeBuffer = millis();                          //Store current time
  while (millis() - timeBuffer < 12000) {         //12 second loop
    if (millis() - timeBuffer < 10000) date = 1;//Display time for 10 seconds
    else date = 0;                                //Show date for remaining 2 seconds
    getSetBright();                               //Get and set brightness
    writeBuffer();                                //Get time from RTC
    for (uint8_t dig = 0; dig < 4; dig++) {       //Light digits, set colon + decimal
      decbit = (~date & (dig == 2)) << 7;         //Turn on/off decimal for date/time
      lightDisp(~(1 << dig) & ~(date << COLON), ~numbers[buffer[dig]] & ~decbit);
    }
  }
}
void getSetBright() {
  DDRB &= ~(1 << clockPin);                  //Set clockPin to input to read sensor
  uint8_t bright = analogRead(BRIGHT) >> 2; //Get brightness and scale to 8-bit
  analogWrite(OE, BOV - bright);             //Write brightness, with blackout value
  DDRB |= 1 << clockPin;                     //Set clockPin to output for shiftOut
}
void lightDisp(uint8_t anode, uint8_t cathode) {
  PORTB &= ~(1 << latchPin);                      //Pull latch low
  shiftOut(dataPin, clockPin, LSBFIRST, anode);   //Shift out anodes
  shiftOut(dataPin, clockPin, LSBFIRST, cathode); //Shift out cathodes
  PORTB |= 1 << latchPin;                         //Set latch high
}
void writeBuffer() {
  TinyWireM.beginTransmission(RTCADDRESS);    //Begin I2C communication with RTC
  if (date) TinyWireM.write(1);               //Start reading from Register 1 (time)
  else TinyWireM.write(4);                    //Start reading from Register 4 (date)
  TinyWireM.endTransmission();                //Send end condition
  TinyWireM.requestFrom(RTCADDRESS, 2);       //Tell RTC how much data is required
  while (!TinyWireM.available());             //wait to place the data in buffer
  uint8_t low = BCD2DEC(TinyWireM.read());    //Store register contents to variable
  uint8_t high = BCD2DEC(TinyWireM.read());   //Store register contents to variable
  if (high > 9) buffer[3] = high / 10;        //If double digit, turn on left digit
  else buffer[3] = 10;                        //If single digit, turn off left digit
  buffer[2] =  high % 10;                     //Store 1s of high digits to buffer
  buffer[1] =  low / 10;                      //Store 10s of low digits to buffer
  buffer[0] =  low % 10;                      //Store 1s of low digits to buffer
}
uint8_t BCD2DEC(uint8_t value) {              //Function to convert BCD to decimal
  return (value >> 4) * 10  + (value & 0x0F); //Converts BCD value to decimal
}
```

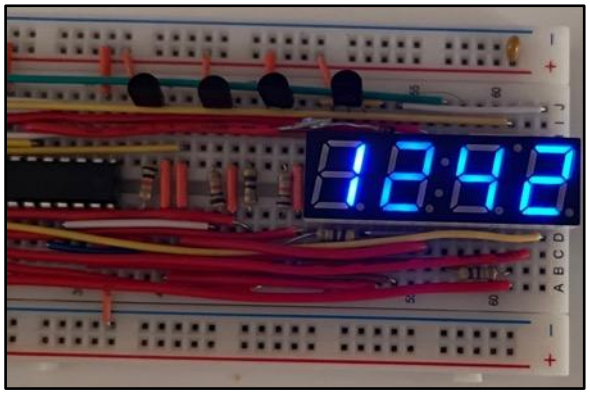
## Media

Project video: https://youtu.be/93xTt9MGoOE

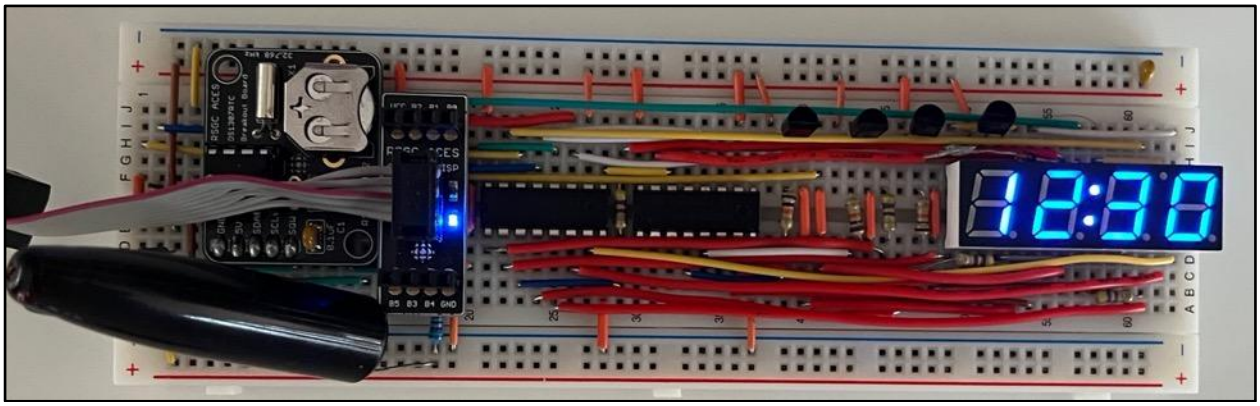


RTC Prototype Circuit Overview

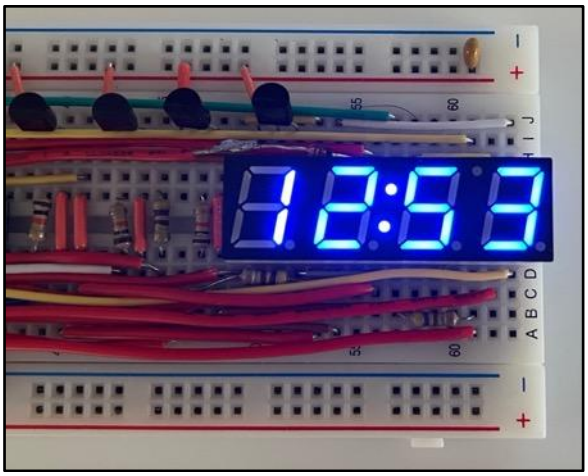
Time 12:42, Light on LDR



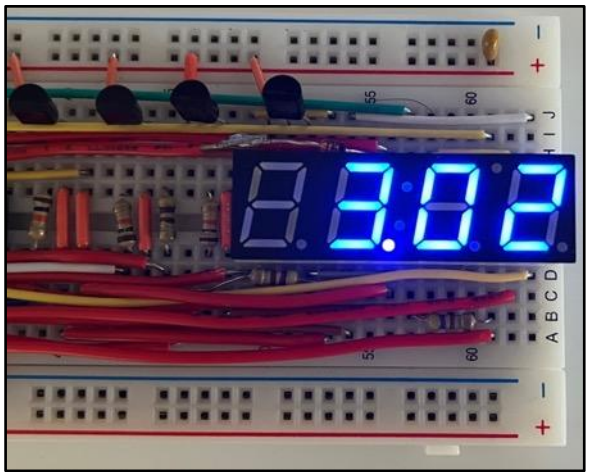
Time 12:42, LDR Covered



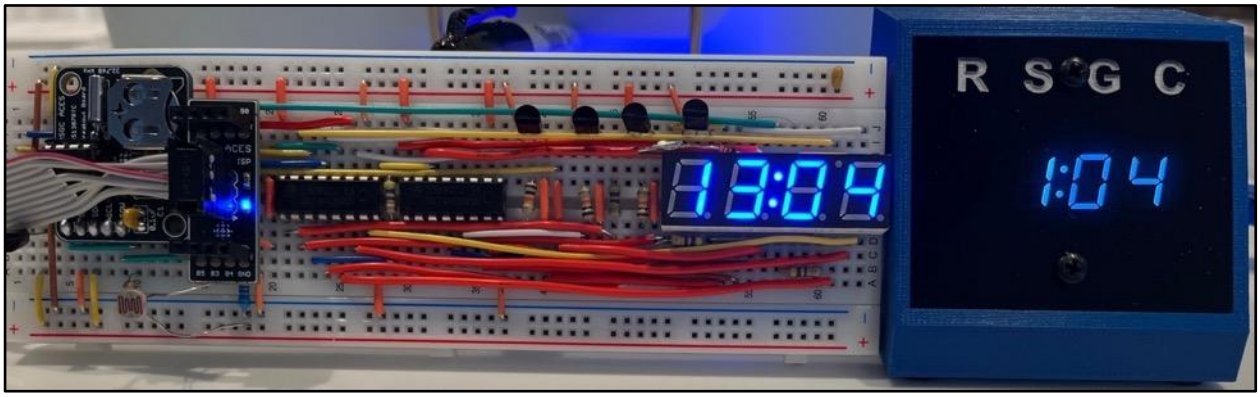
Time 12:30, LDR Covered (Bottom Left)




Time 12:53, LDR Neutral




Date March 2nd, LDR Neutral




Breadboard Prototype and C. D'Arcy's Final Tiny Clock Side-by-Side, Time 1:04 PM


Tiny Clock PCB Inside Case


Tiny Clock Back

Tiny Clock Time 12:59

Tiny Clock March 2nd

## Reflection

If there's anything that I've been reminded of this project, it's that I get carried away too easily. I know it's actually a good thing, in most cases, but it really is true. To me, this project felt like more of an ISP than a required project; every time I got a new idea, I would add it to the clock, to the point where my clock is quite different than the project description. I think that this has been a great project, potentially even the best this year (other than my short ISP). I know that this is a new project, and I hope for there to be more projects with this much freedom in the future.

For some weird reason, I feel like this project marks a new era for me. For example, today, I was looking back at my code writing skills during the binary game project (see Project 2.4), and I saw that one of the statements (which I thought was pretty advanced at the time) could have been easily replaced with a clever for loop. During my short ISP, my nearly two pages of code could have been boiled down to half a page if I had used shift registers and interrupts. I've mentioned it in previous reflections, but the rate at which new concepts are becoming more efficient and useful is astonishing to me.

I also think that the introduction to PCBs is going to be a major game changer. I already have a few ideas of boards to make, like an I2C seven segment display using an ATtiny85, and of course, the few boards I will need for my medium ISP. Overall, like most projects before, while the actual project has been relatively trivial, I have learned a great deal.

# Project 2.7.3 Medium ISP: Autobox

## Purpose

In addition to displaying the current gear or cadence, the purpose the Autobox, which is an automatic bike gear shifter, is to automatically put the rider into the optimal gear, based on how quickly they are pedaling.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI3M/2324/ISPs.html#logs

Project proposal: http://darcy.rsgc.on.ca/ACES/TEI3M/2324/images/RohanMedium.png

## Theory

A servo motor is a rotary actuator, who's angular position can be manipulated by the duty cycle of a PWM signal (see Figure 1). As a result of the PWM control, the servo needs only one pin in addition to power and ground to be controlled. An MCU sends out a PWM signal, allowing the angle to be precisely modulated. In this configuration, a feedback device, usually a potentiometer, is used to detect the position of the servo. A motor with a high gear ratio spins the head, giving it a high amount of torque. The motor spins, until the desired position is detected by the potentiometer.

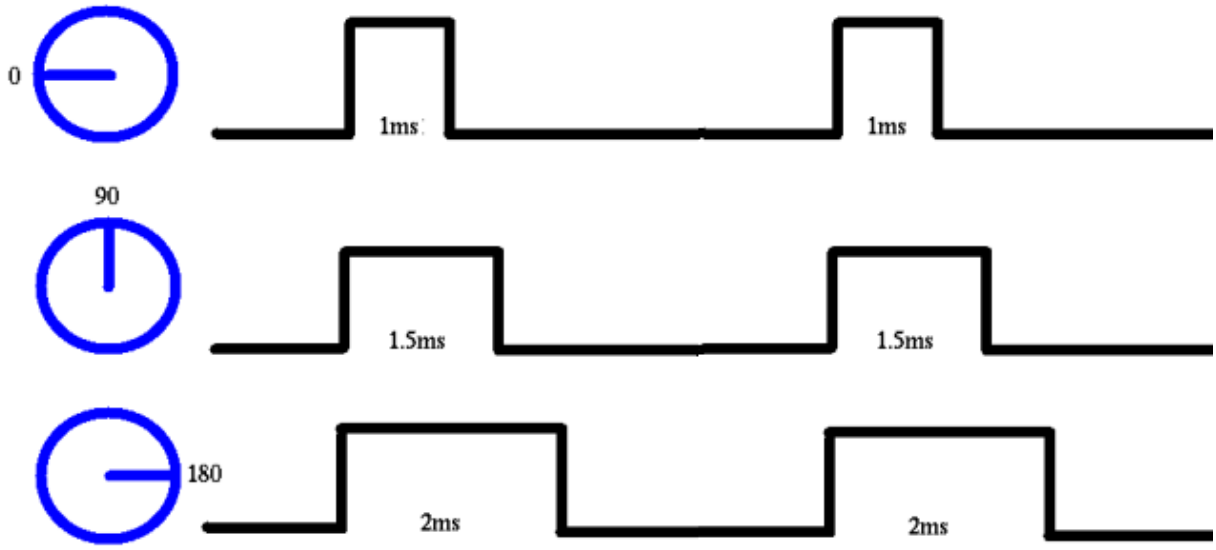



Figure 1. Servo control using PWM duty cycle

I2C is a synchronous protocol that allows different devices to communicate with each other, one byte at a time (see Project 2.6.1). Normally, it is used to communicate between a controller, and a non-processing unit, such as an MCU and RTC (see Project 2.6.2). In some cases, however, it can be used to communicate between multiple MCUs. In this configuration, there is no "master." While there is still one device that is designated as the master, each MCU is primary. This setup has many advantages; it allows multiple processes to run at the same time, and it can take stress off the main MCU to free up processing power for other tasks. It also allows for modular designs; each major section of a machine can have its own MCU that performs a simple task, reporting back to the main MCU to perform a larger, more complex task. Instead of one MCU taking input, processing it, and actuating an output with the data, one MCU can monitor each group of inputs, and send streamlined data to another MCU. Then, the data can be compiled, and sent to the appropriate output. In this configuration, the machine operates as a network, greatly increasing its efficiency and precision.

## Procedure

The Autobox has three main sections: the buttons input unit, the main board, and the display. The main board is housed with the servo motor in a box mounted on the bike frame. It is setup as the master; however, all three devices are considered "primary" (see Theory section). Each device is capable of keeping track of metrics, and performing a complex action.

The buttons unit keeps track of the count, bound at 0 and 5, from the up and down buttons. It sends the current count using the 3 least significant bits, and the state of the switch on bit 7 (see Figure 1).

The main board receives the byte from the buttons device, and chooses manual or automatic mode based on bit 7 (see Figure 1). In manual mode, the gear is adjusted to the gear stored in the count from the buttons device. In automatic mode, the gear is dynamically adjusted based on the rider's cadence.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | ATtiny84 DIP MCU |
| 2 | ATtiny85 DIP MCU |
| 1 | SN74HC595N Shift Register |
| 1 | TPIC6C595 Shift Register |
| 1 | 4 Digit Seven Segment Display |
| 1 | 16 MHz Crystal Oscillator |
| 2 | 22 pF Ceramic Capacitor |
| 6 | 2 × 1 Terminal Block |
| 3 | 2 × 1 ISP Header |
| 8 | 10 KΩ Fixed Resistor |
| 8 | 330 Ω Fixed Resistor |
| 2 | Push Button Normally Open |
| 1 | SPDT Slide Switch |
| 1 | I2C Display PCB |
| 1 | I2C Buttons PCB |
| 1 | Main PCB |
| 1 | 5V Power Supply |
| ~ | Solder |
| ~ | Wires |

| Figure 1. Buttons Device Bit Assignment | | | | | | | |
|---|---|---|---|---|---|---|---|
| **BIT 7** | **BIT 6** | **BIT 5** | **BIT 4** | **BIT 3** | **BIT 2** | **BIT 1** | **BIT 0** |
| Switch | N/A | N/A | N/A | N/A | Count (A) | Count (B) | Count (C) |

The main board uses a reed switch and magnet system to find cadence (see Project 2.5.3). The onboard ATtiny84 waits for two revolutions of the pedals to go around, and measures the time between to calculate RPMs. If the RPMs are too high, it shifts up a gear. Conversely, if the RPMs are too low, it shifts down a gear. Using this mechanism, the rider is always put in the optimal gear. The Autobox keeps the cadence between 70 and 80 RPMs.

| Figure 2. Display Device Bit Assignment | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Byte** | **BIT 7** | **BIT 6** | **BIT 5** | **BIT 4** | **BIT 3** | **BIT 2** | **BIT 1** | **BIT 0** |
| **1** | Val (G) | Val (H) | Val (I) | Val (J) | Val (K) | Val (L) | Val (M) | Val (N) |
| **2** | N/A | N/A | Val (A) | Val (B) | Val (C) | Val (D) | Val (E) | Val (F) |
| **3** | N/A | N/A | N/A | N/A | N/A | N/A | Dec (A) | Dec (B) |

On the display device, when in automatic mode, the cadence is displayed. In manual mode, the current gear is displayed. Similar to the buttons device, the display is its own device. It uses an ATtiny85, a standard 8 output shift register for the anodes, and a TPIC shift register for the cathodes. There are 3 bytes that must be sent for the display to be updated: 2 for the value to be displayed, and 1 for the decimal position. While there are two extra bits for the decimal in the second bit (see Figure 2), due to the manipulation to use them, it is more efficient to send an extra byte.

When the display board receives the bytes in Figure 2, it puts the first two together, to recreate the 16-bit unsigned integer that will be displayed. Then, it splits the transmitted number into its 4 digits and stores it to a buffer. The display MCU then constantly shifts out the correct values from an array-based segment map, and uses POV to display all 4 digits until the next I2C transmission (see Project 2.6.2). This system allows the main MCU to perform other tasks, while the POV effect continues. Additionally, it eliminates the need for the main MCU to perform resource-heavy division operations to split the number into its digits. In this configuration, the number is simply sent over I2C, and the rest is taken care of by an external processor.

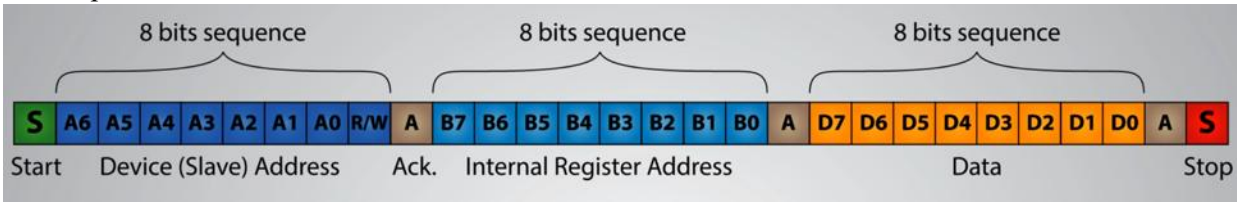



Figure 3. I2C Bus Bit Sequences

When using I2C, there is a "stop condition" that signals the receiving device that the transmission has ended. When creating I2C devices in "slave" mode using an ATtiny85, the stop condition it not automatically detected.; instead, the device must detect it. This is accomplished by running a short loop function, with the line `TinyWireS_stop_check();` executed as many times per second as possible.

As a result of using I2C for the display and buttons, only two pins are utilized, for a total of 4 pins with the cadence sensor and servo motor. While an ATtiny85 has sufficient pins for this application, an ATtiny84 (see Figure 4) makes more sense. The extra pins can be used to accommodate a 16 MHz crystal oscillator, which is necessary to keep precise time. Without the oscillator, the time kept is not as accurate, and the calculated cadence would be inaccurate. The ATtiny84 can accommodate the crystal oscillator which makes it more suitable.

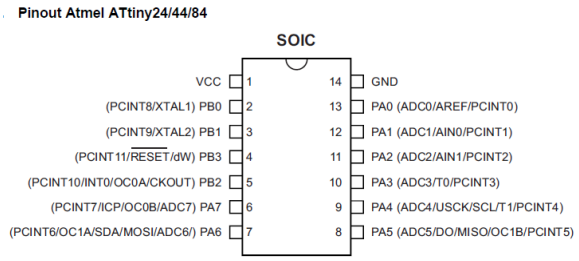



Figure 4. ATtiny84 Pinout

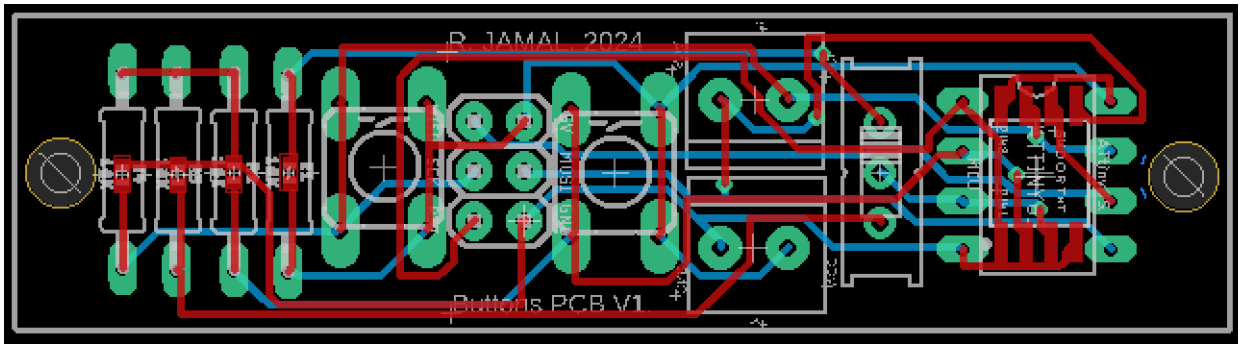



Figure 5. Buttons Device Fusion 360 Board View

Each of the previously mentioned devices feature a PCB, custom-made for the application. The buttons unit features an ISP header, an ATtiny85, 4 resistors, two PBNOs, two terminal blocks, and a SPDT slide switch (see Figure 5). Where applicable, both surface mount technology (SMT) and through hole technology (THT) is included. Since SMT parts are so small, this feature does not increase the footprint.

The main board contains most of the Autobox's components; however, it is still relatively small, with a total size of 25 mm by 16 mm. On the main board are 4 terminal blocks, 1 for I2C communication, 1 for power, and 2 that provide access to I/O pins (see Figure 6). One of these is used to detect the cadence, and the other can be used with a second reed switch system. This would allow the system to have access to the speed, which could be displayed on the seven-segment display. In this configuration, the footprint of the actual display would be relatively small as it would not require a battery onboard.

The board is designed to be a plug and play system; it includes everything necessary to run. All pullup and pulldown resistors are on-board. This includes two 10 KΩ pullup resistors for I2C communication.

Many of the components pictured in Figure 5 are not used in the final design of the Autobox; some components were not necessary. For example, the 7805-voltage regulator is unable to provide enough current for the servo motor, and was replaced with a boost converter and lithium battery. As a result, the two supporting capacitors are also not needed.

The utilization of a PCB allows the footprint of the circuit to be greatly decreased. If it were done on a point-to-point board, the space requirement of the project would be vastly larger.

The final PCB used in the Autobox is the display PCB (see Figure 7). Similar to the buttons PCB, this board contains space for SMT parts in addition to THT parts. This provides flexibility in the application; if the bottom of the board must be relatively flat, SMT parts can be utilized; if the flatness of the board does not matter, THT can be used. The display PCB is essentially a custom, programmable I2C "backpack," that can perform calculations. This is an important feature, as it frees up resources on the other MCUs.

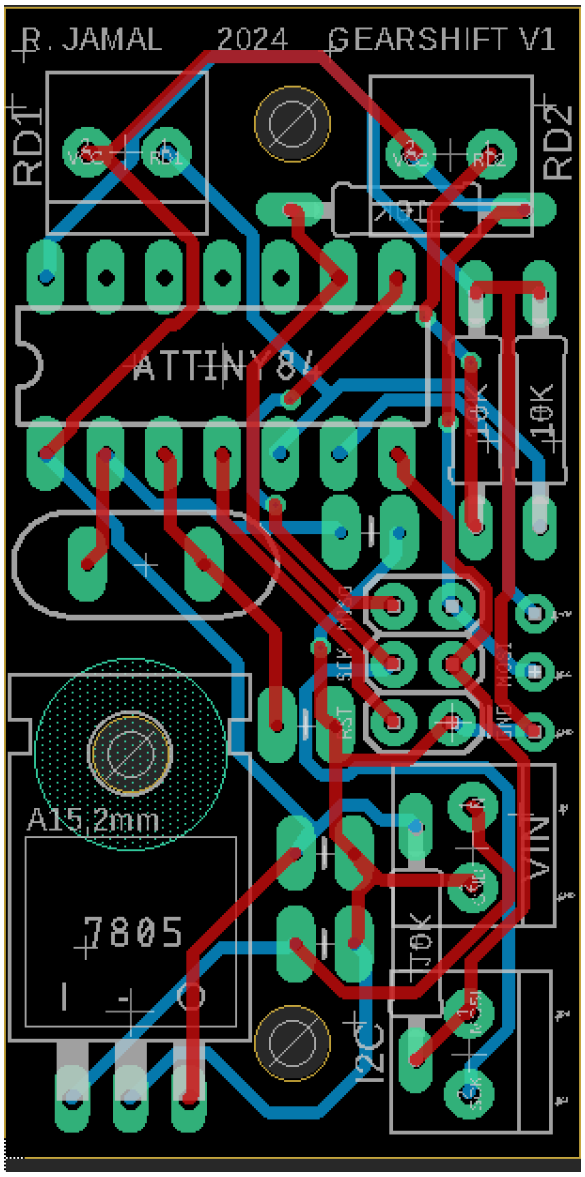



Figure 6. Main PCB Fusion 360 View

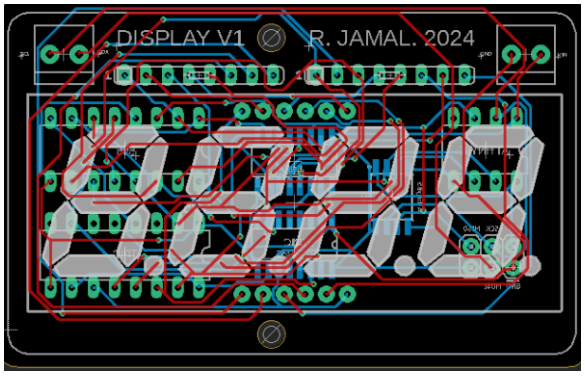



Figure 7. Display PCB Fusion 360 View

# Code

## Main (ATtiny84)

```
// PROJECT    : Bike gear shifter
// AUTHOR     : R. Jamal
// PURPOSE    : To automatically put bike in correct gear using servo
// COURSE     : ICS3U-E
// DATE       : 03 02 2024
// MCU        : TINY84
// STATUS     : [Semi] Working
// NOTES      : Software works, hardware overheated. Written in IDE 1.8.19
#include <Servo_ATTinyCore.h> //Servo library
#include <TinyWireM.h>        //I2C library
#define CAD 8                 //Cadence pin
#define MAX 80                //Max cadence
#define MIN 70                //Min cadence
#define CLDN 20               //Display cooldown time
#define SERVO 7               //Servo pin
#define TOUT 2500             //Timeout (cadence=0)
#define CVRSN 60000           //RPM conversion constant
#define BADDRESS 0x22         //Buttons address
#define DISPADDR 0x30         //Display address
uint8_t buttonState, preCog;  //8-bit unsigned variables
uint8_t cog = 0;              //Current gear variable
uint8_t gearsUp[] = {0, 60, 80, 105, 130, 165}; //Index up
uint8_t gearsDown[] = {0, 25, 42, 60, 80, 165}; //Index down
uint16_t bufferRPM = 1234;    //Buffer to store cadence to
uint64_t lastTrans;           //Keeps track of I2C transmissions
Servo myservo;                //Servo object
void setup() {                //Setup, runs once
  TinyWireM.begin();          //Start I2C communication
  myservo.attach(SERVO);      //Attach servo to servo pin
}
void loop() {                                   //Loop function, runs continously
  buttonState = getButtons();                   //Store number in buttons
  if (buttonState >> 7) {                       //MSB stores mode switch value
    bufferRPM = getReedRPM(CAD);                //Read pedals speed (cadence)
    lightDisplay(bufferRPM, 0);                 //Displays the cadence
    if (cog != 5 && bufferRPM > MAX) cog++;     //Statement to increase gear
    if (cog && bufferRPM && bufferRPM < MIN)    //Statement to decrease gear
      cog--;                                    //Statement to decrease gear
  }
  else {                                        //Sets stored gear when manual
    cog = buttonState;                          //Sets cog to stored gear
    lightDisplay(cog, 0);                       //Display the current gear
  }
  if (cog < preCog)                             //Decide which index to use
    myservo.write(gearsDown[cog]);              //Shift down a gear
  else if (cog > preCog)                        //Decide to use up index
    myservo.write(gearsUp[cog]);                //Shift up a gear
  preCog = cog;                                 //Update previous cog, to monitor
}
uint16_t getReedRPM(uint8_t pin) {                          //Function to get cadence
  uint64_t timeOld = millis();                              //Start timeout timer
  while (!digitalRead(pin) && millis() - timeOld < TOUT); //Wait for reed to go high
  while (digitalRead(pin));                                 //Debounce
  timeOld = millis();                                       //Start timer (for RPMs)
  while (!digitalRead(pin) && millis() - timeOld < TOUT); //Wait for full revolution
  while (digitalRead(pin));                                 //Debounce
  if ((millis() - timeOld) < TOUT)                          //Return RPMs; no timeout
    return CVRSN / (millis() - timeOld);                    //Return RPMs
  else return 0;                                            //Returns 0 if timed out
}
```

```
void lightDisplay(uint16_t val, uint8_t dec) {  //Function to light display
  if ((millis() - lastTrans) > CLDN) {          //Display has 20 ms "cooldown"
    lastTrans = millis();                       //Keeps track of last transmission
    TinyWireM.beginTransmission(DISPADDR);      //Begins transmission to display
    TinyWireM.write(val);                       //Sends first byte
    TinyWireM.write(val >> 8);                  //Sends second byte
    TinyWireM.write(dec);                       //Sends third (decimal) byte
    TinyWireM.endTransmission();                //Ends transmission
  }
}
uint8_t getButtons() {                  //Receives state of buttons from slave
  TinyWireM.requestFrom(BADDRESS, 1); //Requests 1 byte from buttons device
  while (!TinyWireM.available());     //Waits for data to be placed into buffer
  return TinyWireM.read();            //Returns the buffer-stored value
}
```

## Buttons Device (ATtiny85)

```
// PROJECT   : Button device slave (I2C)
// AUTHOR    : R. Jamal
// PURPOSE   : To keep track of buttons and communicate states over I2C
// COURSE    : ICS3U-E
// DATE      : 03 02 2024
// MCU       : TINY85
// STATUS    : Working
// NOTES     : Written in IDE 1.8.19
#include <TinyWireS.h>                          //I2C slave library
#define UP 1                                    //Up button pin
#define DN 3                                    //Down button pin
#define SWTCH 4                                 //Switch pin
#define ADDRESS 0x22                            //I2C address
uint8_t val = 0;                                //Value for +/-
void setup() {                                  //Setup, runs once
  TinyWireS.begin(ADDRESS);                     //Define address
  TinyWireS.onRequest(request_ISR);             //Define request ISR
}
void loop() {                                   //Loop, runs continously
  TinyWireS_stop_check();                       //Detect stop condition
  if (digitalRead(UP) && val != 5) val++;       //Wait for up button
  while(digitalRead(UP));                       //Debounce
  if (digitalRead(DN) && val != 0) val--;       //Wait for down button
  while(digitalRead(DN));                       //Debounce
}
void request_ISR() {                            //ISR for requests
  TinyWireS.send(val | digitalRead(SWTCH) << 7);  //Send values over I2C
}
```

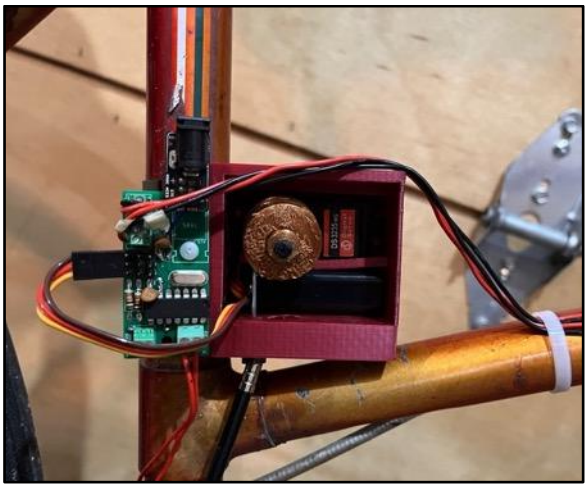
## Display Device (ATtiny85)

```
// PROJECT   : Display device slave (I2C)
// AUTHOR    : R. Jamal
// PURPOSE   : To display value received over I2C on display
// COURSE    : ICS3U-E
// DATE      : 03 02 2024
// MCU       : TINY85
// STATUS    : Working
// NOTES     : Written in IDE 1.8.19
#include <TinyWireS.h>              //I2C slave library
#define clockPin 4                  //Clock pin shift register
#define latchPin 3                  //Latch pin shift register
#define dataPin 1                   //Data pin shift register
#define ADDRESS 0x30                //I2C address
uint16_t rec;                       //Receieved data buffer
uint8_t dec;                        //Decimal data buffer
bool de;                            //Decimal enabled bool
uint8_t numbers[] = {               //Start segment map
  B11111100,                        //0 (a,b,c,d,e,f)
  B01100000,                        //1 (b,c)
  B11011010,                        //2 (a,b,d,e,g)
  B11110010,                        //3 (a,b,c,d,g)
  B01100110,                        //4 (b,c,f,g)
  B10110110,                        //5 (a,c,d,f,g)
  B10111110,                        //6 (a,c,d,e,f,g)
  B11100000,                        //7 (a,b,c)
  B11111110,                        //8 (a,b,c,d,e,f,g)
  B11110110,                        //9 (a,b,c,d,f,g)
};                                  //End segment map
uint8_t buffer[] = {4, 3, 2, 1};    //Intermediate data holder
void setup() {                      //Setup function, runs once
  TinyWireS.begin(ADDRESS);         //Define I2C address
  pinMode(dataPin, OUTPUT);         //Set dataPin as output (DDR)
  pinMode(latchPin, OUTPUT);        //Set latchPin as output (DDR)
  pinMode(clockPin, OUTPUT);        //Set clockPin as output (DDR)
}
void loop() {                               //Loop function, runs continuously
  TinyWireS_stop_check();                   //Detect stop condition
  writeBuffer();                            //Store received number to buffer
  for (uint8_t dig = 4; dig < 8; dig++) {   //Light digits, set decimal
    if ((dig - dec) == 3) de = 1;           //Decide when to enable decimal
    else de = 0;                            //Logic for decimal point
    lightDisp(1 << dig, numbers[buffer[dig - 4]] | de); //Shiftout segments
  }
}
void lightDisp(uint8_t anode, uint8_t cathode) {  //Function to shiftout segments
  PORTB &= ~(1 << latchPin);                      //Pull latch low
  shiftOut(dataPin, clockPin, LSBFIRST, anode);   //Shift out anodes
  shiftOut(dataPin, clockPin, LSBFIRST, cathode); //Shift out cathodes
  PORTB |= 1 << latchPin;                         //Set latch high
}
void writeBuffer() {                  //Function, stores received value to buffer
  if (TinyWireS.available()) {        //Waits until there is data being sent
    rec = TinyWireS.receive();        //Receives first byte
    rec |= TinyWireS.receive() << 8;  //Receives second byte
    dec = TinyWireS.receive();        //Receives decimal byte
  }
  buffer[3] = rec / 1000;             //Gets thousands digit
  buffer[2] = (rec / 100) % 10;       //Gets hundreds digit
  buffer[1] = (rec / 10) % 10;        //Gets tens digit
  buffer[0] = rec % 10;               //Gets onces digit
}
```

## Media

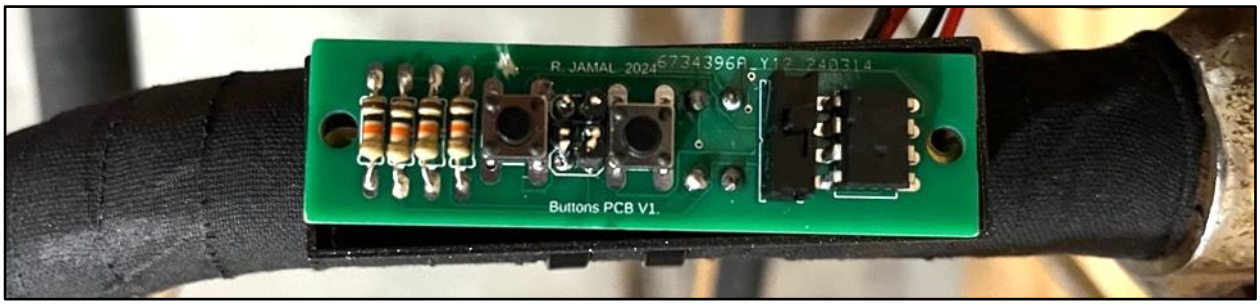
Project video: https://youtu.be/KcbHWWFukyQ

Autobox Mounted on Bike, Side Profile



Autobox, Closeup

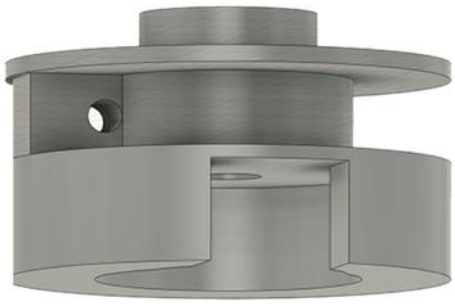
Buttons



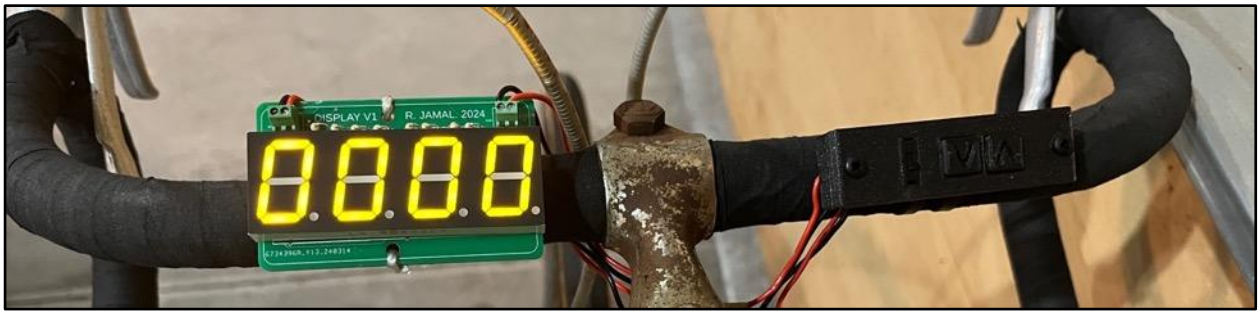

Buttons PCB Mounted on Handlebars

Buttons PCB Mounted on Handlebars with Cover


Display PCB Closeup



Servo Horn Attachment




Handlebars View

## Reflection

This ISP has been a complete rollercoaster for me. Certain parts of the circuit worked really well right away, but others took a lot of fiddling with to get working. For example, when I was trying to get my devices to talk to each other over I2C, I spent many hours debugging the code, hardware, and connections. Eventually, I found out that the problem was not with my setup, but the library I was using. That version turned out to be corrupt, so I had to downgrade it. I guess what Max said in his presentation really resonated with me: libraries should be avoided due to their mystery. They're great when they're working, but when something goes wrong, you have no idea where to start to debug it. This is exactly what happened to me. My hardware was right, and the code I had written seemed to be okay. As a result, I spent much time and frustration trying to fix my setup. Had I not stumbled upon a forum post explaining that the most recent version of the TinyWireS library was corrupt, I don't know that I would ever have found out. The biggest problem here with libraries is that you generally have little to no idea what is really happening. Even after I found out that the library was corrupt, I still have no idea what that means. Was the file not readable? Was there a mistake in the library? Something else entirely?

Another lesson learned this ISP is that even if your idea is theoretically a good idea, even if you do manage to get it to work, it may not have a super long lifespan. I had my project mostly working on the Tuesday before the Friday I was presenting. Unfortunately, as I learned, 3d-printed plastic was not the best choice for my project; the servo motor got really hot and started bending the plastic, which made the horn rub against itself. As a result, the servo did not have enough torque to move the cable anymore. The 2 days that the project worked were pretty good, but I definitely wish I would have used a stronger material that didn't insulate heat so well. I'm pretty sure the servo also overheated, because its behaviour was also unpredictable after it got hot.

This part is not really related to the project, more of a reflection on the course, but I'm already seeing the dividends of my work into the course outside of school. So, backstory: almost all the lights in my house are controlled by a centralized computer system/control panel. Unfortunately, the computer stopped working for some reason on Friday, so we couldn't use any of the lights. Eventually, based on testing it in certain scenarios, I figured out that the problem was the power supply. Before this course, I would have just looked at all the wiring and figured it was impossible for me to figure out. But then, I realized if this thing had a datasheet, it would probably have a pinout. Sure enough, I found the pinout and I was able to solve the problem. As a bonus, the control panels and buttons throughout the house communicate using I2C! I think that this experience does, in a way, tie back to ISPs. The reason that we can choose whatever we want for the project with almost no restrictions, is because it does not matter what we choose. It's not what we learn, but the habits we build along the way. Moral of the story: read the datasheet.

# ICS4U-E

# Project 3.1 CharlieClock

## Purpose

In addition to demonstrating the concept of *Charlieplexing* (see Theory section), the purpose of the CharlieClock is to display the time in an analog fashion using 132 of a combination of red, green, and blue LEDs, controlled by 12 digital I/O pins.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/Tasks.html#CharlieClock

Project schematic: https://oshwlab.com/rjamal/charlieclock

## Theory

*Charlieplexing*, named for engineer Charlie Allen, is the process that involves taking advantage of the third, high-impedance state of digital pins to optimize I/O efficiency. While in most applications, pins are either set high or low, in Charlieplexing applications, pins are set as high, low, or input (otherwise known as high-impedance) which prevents current from flowing through the pin in any direction. In this configuration, *n* number of lines can control *t* individual LEDs (see Figure 1), where $t = \frac{n!}{(n-2)!}$ or $t = n \times (n-1)$. One other way to express this is *n* pick 2. The number 2 comes from the fact that each LED requires 2 lines to be controlled. The calculated value of t is equal to the number of 2-term permutations of *n* names, or lines.

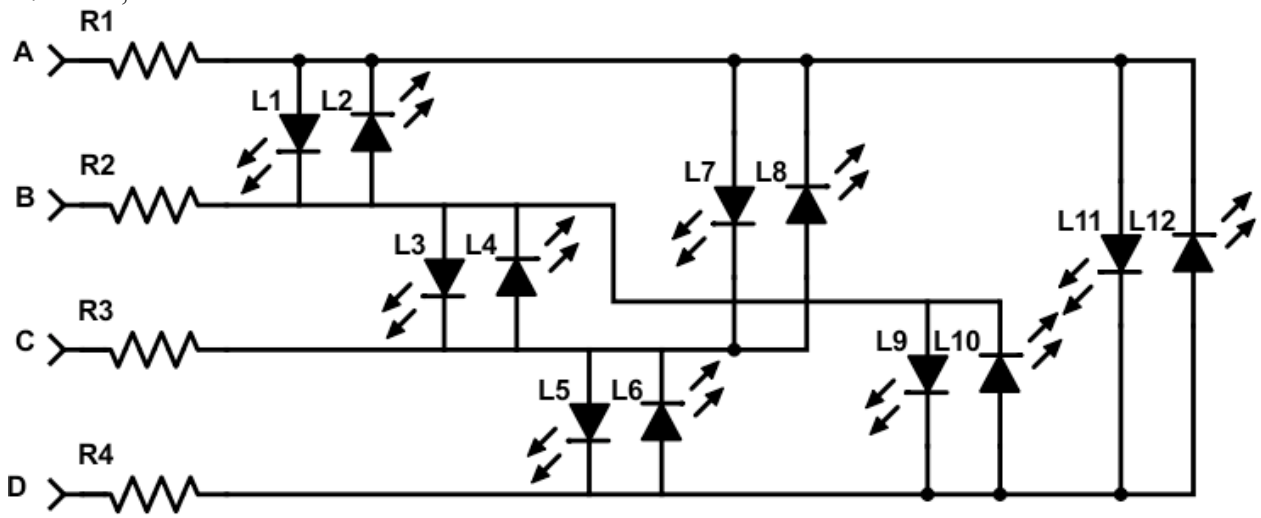



Figure 1. Charlieplexing System with 4 Input Lines and 12 LEDs

Figure 1 depicts 12 LEDs in a Charlieplexed configuration. With 4 lines, t= $4 \times (4-1) = 12$, which means this is the greatest possible number of LEDs in a 4-line system. The advantage of Charlieplexing over traditional LED control is that each LED can be addressed without inadvertently lighting a different one. For example, in Figure 1, if the desired configuration were L1 lit, with all other LEDs not lit, line A would be pulled high, and line B would be pulled to ground. This would cause L1 to light up, but it would also cause L7 and L11 to light up, as current would flow through them from lines A to C and A to D, respectively; with the addition of Charlieplexing, however, lines C and D would be set as inputs, meaning that no current would flow between them and any other line. In other words, only L1 would be lit up.

## Procedure

The CharlieClock consists of 132 LEDs – 1 for each hour, minute, and second – in a Charlieplexed configuration (see Theory section). By chance, 132 is 12 pick 2, so to interface with all 132 LEDs, the CharlieClock utilizes a total of 12 digital I/O pins.

To convert the information stored in the registers of the DS1307 RTC (see Project 2.6.2) to a human-readable time and date on the CharlieClock, an Atmega328P MCU is utilized.

The MCU reads the time from the RTC, then displays the received value on the CharlieClock, as well as the date on a custom seven-segment I2C display (see Project 2.7.3).

The LEDs are arranged in a circle, evenly spaced, allowing them to indicate the time (see Figure 1) in a similar fashion to that of an analog clock with moving hands. In this configuration, the outer ring contains 12 blue LEDs, with each one representing an hour. The middle ring contains 60 green LEDs, with each one representing a minute. Finally, the inner ring contains 60 red LEDs, with each representing a second.

On its own, the CharlieClock board contains no logic or controlling circuitry. It is simply 132 surface mounted LEDs arranged in a clock-like pattern, wired together in a Charlieplexed configuration. Access to the 12 control lines is provided via soldered header pins (see Figure 1).

These 12 lines connect to the motherboard PCB (see Figure 2). This PCB contains the logic that powers the clock, including the RTC, and MCU. Rather than fetch the time from the RTC constantly, the MCU simply fetches the time once over I2C, then adds 1 second for every 2 ticks of a 1 Hz pin-change interrupt, driven by the square wave pin of the RTC. This is significantly more efficient than the timekeeping used in Project 2.6.2, as it replaces many I2C transmissions per second with 2 interrupts per second, increasing the POV rate.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | ACES CharlieClock PCB |
| 1 | Motherboard PCB |
| 1 | Display PCB |
| 1 | 3D-Printed CharlieClock Stand |
| 1 | Coin cell battery retainer |
| 1 | SMT Atmega328P MCU |
| 1 | SMT DS1307 RTC |
| 5 | 0.1 µF Capacitor |
| 6 | 10 KΩ Fixed Resistor |
| 1 | 32.768 Hz Crystal Oscillator |
| ~ | Solder Paste |
| ~ | Solder |

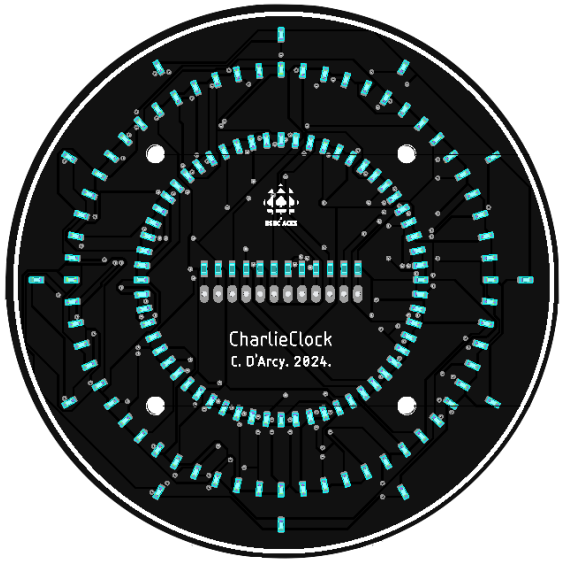



Figure 1. CharlieClock

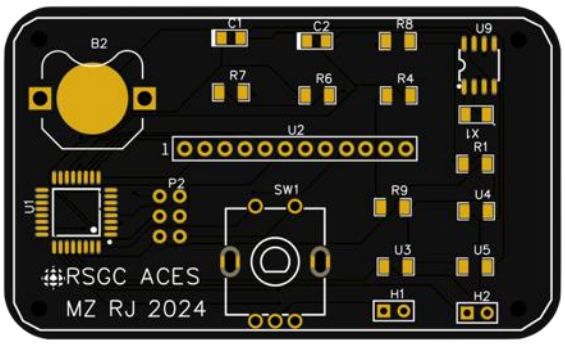



Figure 2. Motherboard PCB

The inclusion of a *rotary encoder* allows the user to change the time with the press of a button and spin of a dial, eliminating the need to set the RTC programmatically. A rotary encoder is a device that converts changes in angular position to digital signals (see Figure 3). Converting the spin of the rotary encoder to an increment or decrement (depending on the direction of its spin) allows the user to spin the device and watch the CharlieClock update to their will.

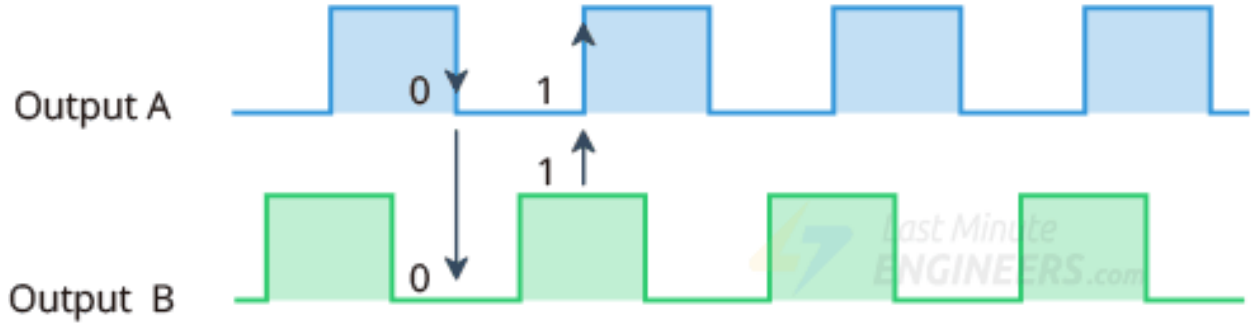



Figure 3. Rotary Encoder Output Characteristics

As the device rotates, the two outputs, A and B, come into contact with a metal plate carrying a high signal (see Figure 4). Depending on which direction the encoder rotates in, either A or B will go high first.

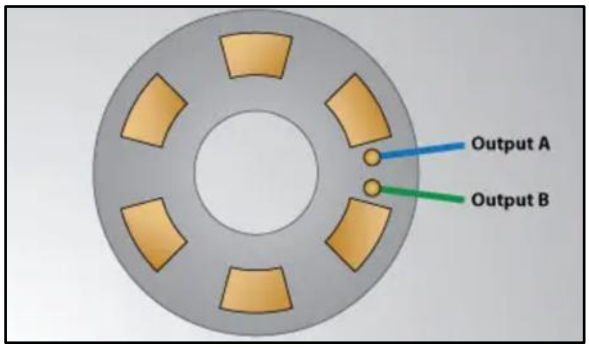



Figure 4. Rotary Encoder Contacts

The output signal depicted in Figure 3 can be interpreted as a counter-clockwise motion because output B goes high before output A does, which Figure 4 shows as a counter-clockwise motion

To convert the desired number to the correct LED on the CharlieClock, a *struct* is used as a pin map. A struct, short for structure, is a collection of different datatypes, in this case, the cathode and anode pin of each LED. To light an LED, the program sets all used pins to inputs, or high-impedance (see Theory section). Then, referring the the struct pin map, it sets the anode and cathode to outputs. While slightly inefficient, this ensures that only the anode and cathode pins will allow current to flow, preventing undesired lighting of LEDs. Finally, it sets the anode high, and the cathode low.

Figure 5 depicts the CharlieClock case attached to its 3-part stand (each part friction-fits together). In this configuration (though not depicted in Figure 5), there is a rotating bezel around the circular part of the case that connects to the rotary encoder, allowing the user to change the time.

The date display fits into the base of the stand (see Figure 5). Additionally, the bottom of the base features a cut-out strip for wires, which connects to a hole in the centre of the vertical rod which allows wires to be routed entirely internally.

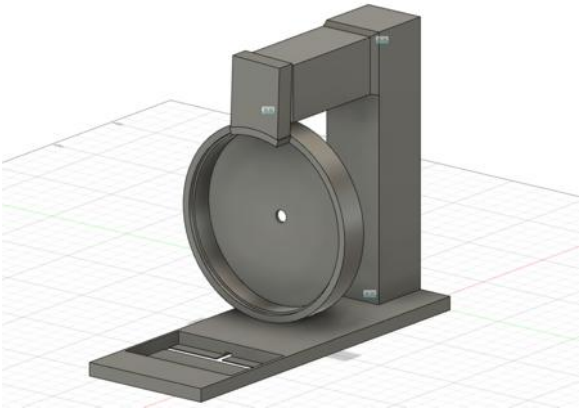


Figure 5. CharlieClock Case and Stand

The motherboard PCB utilizes surface mount parts. Unlike traditional through hole parts, these parts only occupy real estate on 1 side. While this makes them smaller and therefore more space-efficient, they are also more difficult to solder, as there are no holes.

To solder these parts, a surface mount stencil is used (see Figure 6). In this configuration, the stencil is lined up with the PCB, and solder paste is applied. The stencil is cutout in such a fashion that only the traces for which solder is desired allow the paste to pass through.

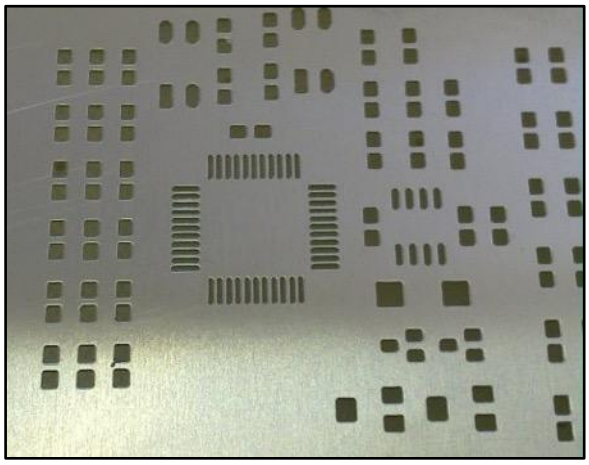


Figure 6. Surface Mount Stencil

Solder paste is made up of very small pieces of solder and flux. This allows it to be spread while cold. When heated up, the flux evaporates, and the solder solidifies. Once the solder paste is applied to the board, the stencil is removed, and the components are placed. Finally, the board and components are transferred to a reflow oven.

A reflow oven is the alternative to the solder iron when using many surface mount parts. It heats the board, according to a *thermal profile* (see Figure 7). A thermal profile is a set data that specifies timing and temperatures, for optimal solder joints.

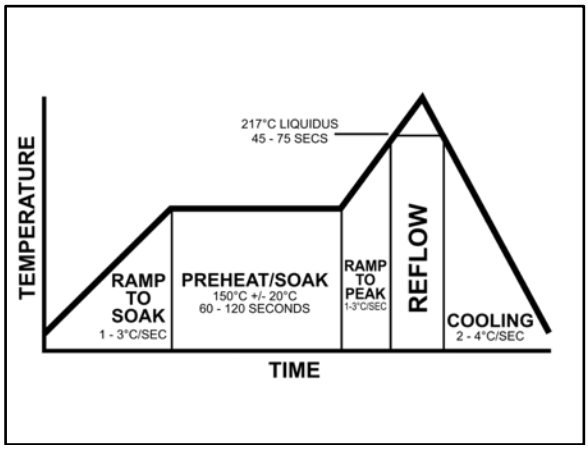



Figure 7. Reflow Oven Sequence

First, the temperature climbs, at a rate between 1 and 3 degrees per second until it reaches 130 to 170 degrees. At this point is known as the soak phase (see Figure 7). During this time, the board slowly reaches a higher temperature, and the flux starts *oxide reduction*. Oxide reduction is the process that prevents metals from oxidizing as oxidized metals have poor conductive properties.

After 1 to 2 minutes, the temperature climbs steadily, until it reaches a peak above the liquid state of solder. During this duration, the solder paste melts, and bonds with the pads. Even if the paste or components are not perfectly aligned with the board traces, as the flux starts to activate, the components align themselves, provided they are within 0.25 mm of their desired trace.

## Code

### Main (ATmega328P)

```
// PROJECT    : CharlieClock
// AUTHOR     : R. Jamal and M. Zischka
// PURPOSE    : To display time on CharlieClock
// COURSE     : ICS4U-E
// DATE       : 04 10 2024
// MCU        : ATMEGA328P
// STATUS     : Working
// NOTES      : Written in 1.8.19
#include  <Wire.h>
#define P1 17
#define P2 16
#define P3 15
#define P4 14
#define P5 13
#define P6 12
#define P7 11
#define P8 8
#define P9 7
#define P10 6
#define P11 5
#define P12 4
#define DS1307RTCADDRESS  0x68      //RTC I2C Address
#define DISPLAYADDRESS    0x30      //Display I2C Address
#define SW 2                        //Rotary encoder button pin
#define PINA 3                      //Rotary encoder pin 'A'
#define PINB 10                     //Rotary encoder pin 'B'
uint8_t pins[] = {P1, P2, P3, P4, P5, P6, P7, P8, P9, P10, P11, P12};
uint8_t numPins = sizeof(pins);     //Number of charlieplexed pins
uint8_t preA, date, mon;             //Last state of RE 'A' pin
uint8_t count = 0;                  //Which attribute encoder affects
volatile uint8_t div2 = 0;          //Variable to divide RTC PCINT by 2
volatile uint8_t pin_A, pin_B;      //Current states of pins A and B
volatile bool changed = false;      //Boolean value for rotary encoder changes
volatile int16_t time[] = {0, 0, 0};//Array to store the current time
int64_t lastTrans = -10000;
struct LED {                        //Struct to hold LED pin maps
  uint16_t cathode;                 //Cathode
  uint8_t anode;                    //Anode
};
LED hours[] = {                     //Hours pin map
  {P1, P9}, {P1, P10}, {P1, P11}, {P1, P12}, {P11, P12}, {P1, P5}, {P1, P4},
  {P1, P3}, {P1, P2}, {P1, P6}, {P1, P7}, {P1, P8}
};
LED minutes[] = {                   //Minutes pin map
  {P2, P8}, {P2, P9}, {P2, P10}, {P2, P11}, {P2, P12},
  {P12, P1}, {P12, P2}, {P12, P3}, {P12, P4}, {P12, P5},
  {P12, P6}, {P12, P7}, {P12, P8}, {P12, P9}, {P12, P10}, {P12, P11},
  {P11, P10}, {P11, P9}, {P11, P8}, {P11, P7}, {P11, P6}, {P11, P5},
  {P11, P4}, {P11, P3}, {P11, P2}, {P11, P1}, {P10, P12}, {P10, P11},
  {P10, P9}, {P10, P8}, {P10, P7}, {P10, P6}, {P10, P5},
  {P10, P4}, {P10, P3}, {P10, P2}, {P10, P1}, {P4, P7}, {P4, P6}, {P4, P5},
  {P4, P3}, {P4, P2}, {P4, P1}, {P3, P1}, {P3, P2}, {P3, P4}, {P3, P5},
  {P3, P6}, {P3, P7}, {P3, P8}, {P3, P9}, {P3, P10}, {P3, P11}, {P3, P12},
  {P2, P1}, {P2, P3}, {P2, P4}, {P2, P5}, {P2, P6}, {P2, P7}
};
LED seconds[] = {                   //Seconds pin map
  {P5, P1}, {P5, P2}, {P5, P3}, {P5, P4}, {P5, P6}, {P5, P7}, {P5, P8},
  {P5, P9}, {P5, P10}, {P5, P11}, {P5, P12}, {P4, P8}, {P4, P9}, {P4, P10},
  {P4, P11}, {P4, P12}, {P9, P12}, {P9, P11}, {P9, P10}, {P9, P8}, {P9, P7},
  {P9, P6}, {P9, P5}, {P9, P4}, {P9, P3}, {P9, P2}, {P9, P1}, {P8, P12},
```

```
  {P8, P11}, {P8, P10}, {P8, P9}, {P8, P7}, {P8, P6}, {P8, P5}, {P8, P4},
  {P8, P3}, {P8, P2}, {P8, P1}, {P7, P12}, {P7, P11}, {P7, P10}, {P7, P9},
  {P7, P8}, {P7, P6}, {P7, P5}, {P7, P4}, {P7, P3}, {P7, P2}, {P7, P1}, {P6, P1},
  {P6, P2}, {P6, P3}, {P6, P4}, {P6, P5}, {P6, P7}, {P6, P8}, {P6, P9},
  {P6, P10}, {P6, P11}, {P6, P12}
};
uint8_t numHours = sizeof(hours) / sizeof(LED);     //Total number of hours LEDs
uint8_t numMins = sizeof(minutes) / sizeof(LED);    //Total number of minutes LEDs
uint8_t numSecs = sizeof(seconds) / sizeof(LED);    //Total number of seconds LEDs

void setup() {
  attachInterrupt(digitalPinToInterrupt(SW), ISR_SWPRESSED, RISING);  //Encoder INT
  cli();                                            //Clear interrupts/noInterrupts();
  PCICR |= (1 << PCIE0);                            //Enable PCINT for pins D8-D13
  PCMSK0 |= (1 << PCINT1);                          //Mask PCINT for pin 9
  if (count != 0)
    PCMSK1 &= ~(1 << PCINT1);
  sei();                                            //Set interrupts/Interrupts();

  Wire.begin();                                     //Initialize I2C
  Wire.beginTransmission(DS1307RTCADDRESS);         //Start a transmission to RTC
  Wire.write(0);                                    //Start reading from Register 0
  Wire.endTransmission();                           //End I2C transmission
  Wire.requestFrom(DS1307RTCADDRESS, 6);            //Specify how much data required
  while (!Wire.available());                        //Wait data to enter read buffer

  time[0] = BCD2DEC(Wire.read());                   //Store first byte to sec buffer
  time[1] = BCD2DEC(Wire.read());                   //Store second byte to min buffer
  time[2] = BCD2DEC(Wire.read());                   //Store third byte to hours buffer
  date = Wire.read();                               //Discard fourth byte
  date = BCD2DEC(Wire.read());                      //Store fifth byte (overwriting 4th)
  mon = BCD2DEC(Wire.read());                       //Store sixth byte to month buffer
}

void loop() {
  if ((millis() - lastTrans) > 9999) {              //Wait 9999 ms - avoid overloading
disp
    Wire.beginTransmission(DISPLAYADDRESS);         //Begin transmission with display
    Wire.write(mon);                                //Pass the month to the display MCU
    Wire.write(date);                               //Pass the date to the display MCU
    Wire.write(3);                                  //Store the decimal in the 3rd byte
    Wire.endTransmission();                         //End the transmission
  }
  if (count)                                                //When count != 0 (set mode)
    attachInterrupt(digitalPinToInterrupt(PINA), ISR_ROTARYCHANGE, CHANGE); //Attach
  else detachInterrupt(digitalPinToInterrupt(PINA));  //If count = 0 detatch
  if (changed) {                                    //When change flag has tripped
    changed = false;                                //Reset change flag
    if (pin_A != preA) {                            //Determine direction
      preA = pin_A;                                 //Store current state of pin_A
      if (pin_A == pin_B) time[count - 1]--;        //Decrement time (direction A)
      else time[count - 1]++;                       //Increment time (direction B)
    }
  }
  if (time[0] == 60) {  //Seconds wraparound (positive)
    time[1]++;
    time[0] = 0;
  }
  if (time[0] < 0) {    //Seconds wraparound (negative)
    time[1]--;
    time[0] = 59;
  }
  if (time[1] == 60) {  //Minutes wraparound (positive)
```

```
    time[2]++;
    time[1] = 0;
  }
  if (time[1] < 0) {    //Minutes wraparound (negative)
    time[2]--;
    time[1] = 59;
  }

  for (uint8_t pin = 0; pin < numPins; pin++)       //Set all pins to high-impedance
    pinMode(pins[pin], INPUT);                      //Set all pins to high-impedance
  pinMode(seconds[time[0]].anode, OUTPUT);          //Set seconds anode to output
  pinMode(seconds[time[0]].cathode, OUTPUT);        //Set seconds cathode to output
  digitalWrite(seconds[time[0]].anode, HIGH);       //Set seconds anode high
  digitalWrite(seconds[time[0]].cathode, LOW);      //Set seconds cathode low

  for (uint8_t pin = 0; pin < numPins; pin++)       //Set all pins to high-impedance
    pinMode(pins[pin], INPUT);                      //Set all pins to high-impedance
  pinMode(minutes[time[1]].anode, OUTPUT);          //Set minutes anode to output
  pinMode(minutes[time[1]].cathode, OUTPUT);        //Set minutes cathode to output
  digitalWrite(minutes[time[1]].anode, HIGH);       //Set minutes anode high
  digitalWrite(minutes[time[1]].cathode, LOW);      //Set minutes cathode low

  for (uint8_t pin = 0; pin < numPins; pin++)       //Set all pins to high-impedance
    pinMode(pins[pin], INPUT);                      //Set all pins to high-impedance
  pinMode(hours[time[2] % 12].anode, OUTPUT);       //Set hours anode to output
  pinMode(hours[time[2] % 12].cathode, OUTPUT);     //Set hours cathode to output
  digitalWrite(hours[time[2] % 12].anode, HIGH);    //Set hours anode high
  digitalWrite(hours[time[2] % 12].cathode, LOW);   //Set hours cathode low
}
ISR(PCINT0_vect) {                                  //PCINT attached to RTC SQW pin
  div2++;                                           //Divide frequency to 1 Hz
  if (div2 % 2) time[0]++;                          //Increment seconds on falling edge
}
void ISR_SWPRESSED() {                              //ISR for when button is pressed
  count++;                                          //Increment count
  if (count == 4) count = 0;                        //Count wraparound
}
void ISR_ROTARYCHANGE() {                           //Rotary encoder ISR
  changed = true;                                   //Activate 'changed' flag
  pin_A = digitalRead(PINA);                        //Store state of pin 'A'
  pin_B = digitalRead(PINB);                        //Store state of pin 'B'
}
uint8_t BCD2DEC(uint8_t value) {                    //Function to convert BCD to decimal
  return (value >> 4) * 10  + (value & 0x0F);
}
```

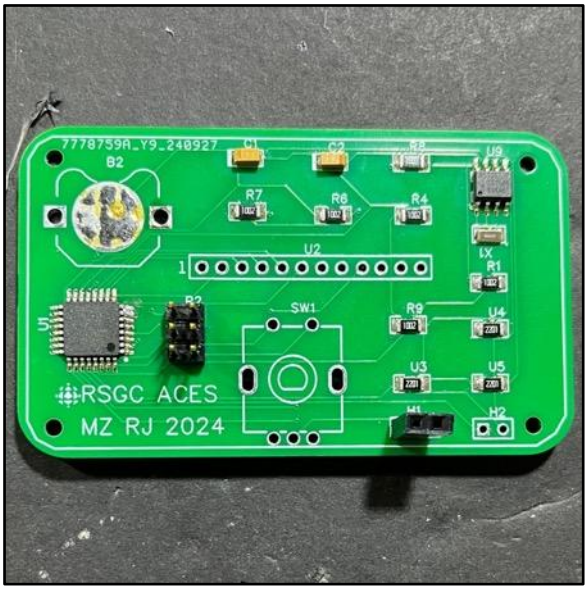
## Display (ATtiny85)

```
// PROJECT    : Display device slave (I2C)
// AUTHOR     : R. Jamal
// PURPOSE    : To display value received over I2C on display
// COURSE     : ICS4U-E
// DATE       : 04 10 2024
// MCU        : TINY85
// STATUS     : Working
#include <TinyWireS.h>                //I2C slave library
#include <TinyWireM.h>                //I2C library for compatibility
#define clockPin 4                    //Clock pin shift register
#define latchPin 3                    //Latch pin shift register
#define dataPin 1                     //Data pin shift register
#define ADDRESS 0x30                  //I2C address
Uint8_t rec1, rec2;                   //Received data buffer
uint8_t dec = 0;                      //Decimal data buffer
bool de;                              //Decimal enabled bool
uint8_t numbers[] = {                 //Start segment map
  B11111100,                          //0 (a,b,c,d,e,f)
  B01100000,                          //1 (b,c)
  B11011010,                          //2 (a,b,d,e,g)
  B11110010,                          //3 (a,b,c,d,g)
  B01100110,                          //4 (b,c,f,g)
  B10110110,                          //5 (a,c,d,f,g)
  B10111110,                          //6 (a,c,d,e,f,g)
  B11100000,                          //7 (a,b,c)
  B11111110,                          //8 (a,b,c,d,e,f,g)
  B11110110,                          //9 (a,b,c,d,f,g)
};                                    //End segment map
uint8_t buffer[] = {4, 3, 2, 1};      //Intermediate data holder
void setup() {                        //Setup function, runs once
  TinyWireS.begin(ADDRESS);           //Define I2C address
  pinMode(dataPin, OUTPUT);           //Set dataPin as output (DDR)
  pinMode(latchPin, OUTPUT);          //Set latchPin as output (DDR)
  pinMode(clockPin, OUTPUT);          //Set clockPin as output (DDR)
}
void loop() {                                 //Loop function, runs continuously
  writeBuffer();                              //Store received number to buffer
  TinyWireS_stop_check();                     //Detect stop condition
  for (uint8_t dig = 4; dig < 8; dig++) {     //Light digits, set decimal
    if ((dig - dec) == 3) de = 1;             //Decide when to enable decimal
    else de = 0;                              //Logic for decimal point
    lightDisp(1 << dig, numbers[buffer[dig - 4]] | de); //Shiftout segments
  }
}
void lightDisp(uint8_t anode, uint8_t cathode) {  //Function to shiftout segments
  PORTB &= ~(1 << latchPin);                      //Pull latch low
  shiftOut(dataPin, clockPin, LSBFIRST, anode);   //Shift out anodes
  shiftOut(dataPin, clockPin, LSBFIRST, cathode); //Shift out cathodes
  PORTB |= 1 << latchPin;                         //Set latch high
}
void writeBuffer() {                    //Function, stores received value to buffer
  if (TinyWireS.available()) {          //Waits until there is data being sent
    rec1 = TinyWireS.receive();         //Receives first byte
    rec2 = TinyWireS.receive();         //Receives second byte
    dec = TinyWireS.receive();          //Receives decimal byte
  }
  buffer[3] = rec1 / 10;                //Gets thousands digit
  buffer[2] = rec1 % 10;                //Gets hundreds digit
  buffer[1] = rec2 / 10;                //Gets tens digit
  buffer[0] = rec2 % 10;                //Gets ones digit
}
```

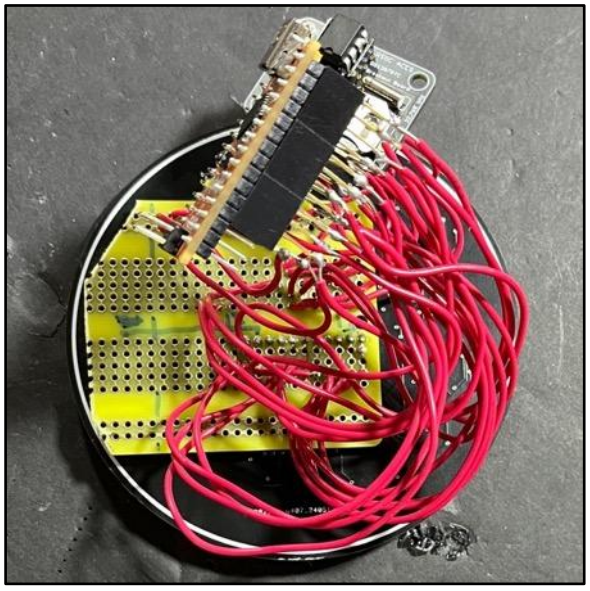
## Media

Project video: https://youtu.be/6-jRYRIcvME

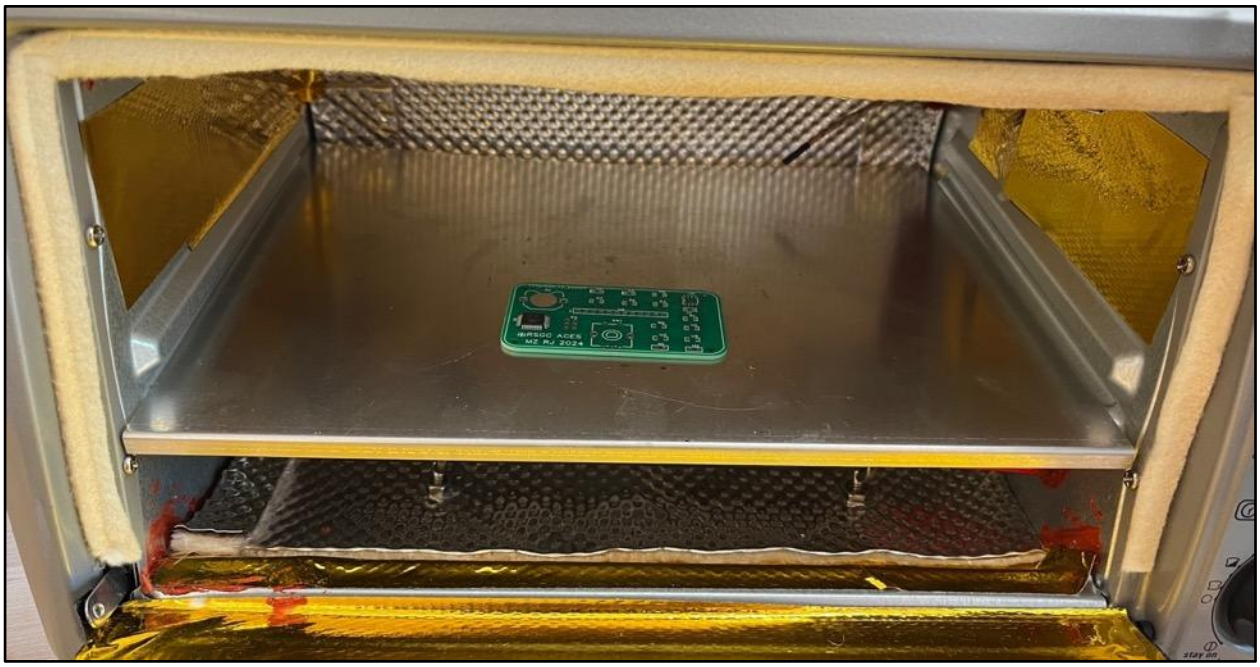

Solder Joints on Atmega328P on The Motherboard




Motherboard PCB



Permaproto Motherboard PCB



Motherboard PCB with Atmega328P in Reflow Oven


CharlieClock on Stand

## Reflection

This has been quite a project. From its start at the end of the last school year, to its end at the start of this school year, it has been through many iterations. Spanning many hours in the DES, quite a few hours at my home, even more at Max's, and multiple all-nighters, I learned so much.

This project really put my CAD skills to the test. I learned EasyEDA (which was true to its name), and dramatically honed my fusion skills. Prior to this project, I didn't really use the sketch feature, but over this project, I've found that it's a literal game changer. The complexity at which I can design has increased tenfold just from the newfound sketch feature

As far as efficiency goes, ever since grade 9, I have found that I work the best when I get into a sort of flow state. When the clock strikes around 1 in the morning, I am so in the zone that I don't think about anything else. It's not even poor time management that leads to this phenomenon; it's just what works best for me. I work better at night.

Last but not least, perhaps even most important, was the collaboration on this project. I learned that when you find someone you work well with, the gains in productivity are more than the sum of your individual talents; I think that Max and I accomplished way more on this project than either one of us could have individually achieved. From baking our parts in the oven for the first time, to having a tightly-integrated case, it was a spectacular build, which I think can largely be accredited to collaboration.

Overall, it was a great project, and I aim for it to set to the tone for the rest of the year.

At least, that is the reflection I had written 2 days ago. Since then, most of the project has had unforeseen errors. Unfortunately, our PCB did not arrive until the Friday, since UPS got delayed. That meant that we had very little turn around if our PCB failed, which it did. We think it was the lack of a 16 MHz crystal, but for whatever reason, we couldn't get our PCB to program.

So, on the Saturday, we rebuilt the PCB using a permaproto board, and a ton of jumper wires (as detailed in Media). I guess that is the thing about these projects: you have to be able to adapt. We ran into many 3D print errors (after failing to realize that we just had to do a bed level), with the final print finishing at 11:14 P.M (yikes!). I'd like to think that whatever was thrown our way, we could come back, but I have to say, it was extremely close. That being said, contrary to what I said earlier, it was a great project, but I really hope that it does not set the tone for this year (at least the time management aspect). In total, hours upon hours were spent, many of which could have been avoided or repurposed, but I guess that's the way it goes. If I could go back, there is not a whole lot I would change (maybe I would have added a crystal to the PCB though).

# Project 3.2 CHUMP (Cheap Homemade Understandable Minimal Processor)

## Purpose

The purpose of *CHUMP* (Cheap Homemade Understandable Minimal Processor) is both to demonstrate how a computer works on the lowest level, and compute simple numbers using standard mathematical operations.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html

## Project 3.2.1 Clock and Counter

### Purpose

The purpose of the clock is to provide a common heartbeat for the entire CHUMP system (including the program counter) to run on. The purpose of the program counter is to keep track of the address of the next instruction to be executed within the program EEPROM.

### Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#tasks

Project schematic: https://crcit.net/c/501cf12b72c74e3582b9cab9ff0a19f8

Program counter datasheet: https://www.ti.com/lit/ds/symlink/sn54ls161a.pdf?ts=1728671937945

https://www.build-electronic-circuits.com/jk-flip-flop/

https://www.youtube.com/watch?v=YW-_GkUguMM

### Theory

Clock signals form the backbone of modern technology. They are able to synchronize the actions of different components of a circuit. Synchronous circuits rely on a clock signal, which is a waveform, often a square wave, that oscillates between high and low logic states. On the rising edge, each component of the circuit can advance.

Within the circuit, the clock signal is fed into a flip-flop (see Figure 1). There are several types of flip-flops; for example, Figure 1 depicts a D flip-flop, which has 1 input. When the clock reaches a rising edge, the output of the flip-flop is updated to the current input. When the clock is low, the inputs can no longer affect the output; it is latched. While there are many different kinds of flip-flops with distinct characteristics, they all abide by the same basic principle; the output can change on either a rising or falling edge, depending on the flip-flop, but it is latched when the clock is in any other state. By having a centralized clock signal fed into flip-flop circuits, changes to the circuit's state of logic can only occur on a certain edge of the clock signal and will occur simultaneously.

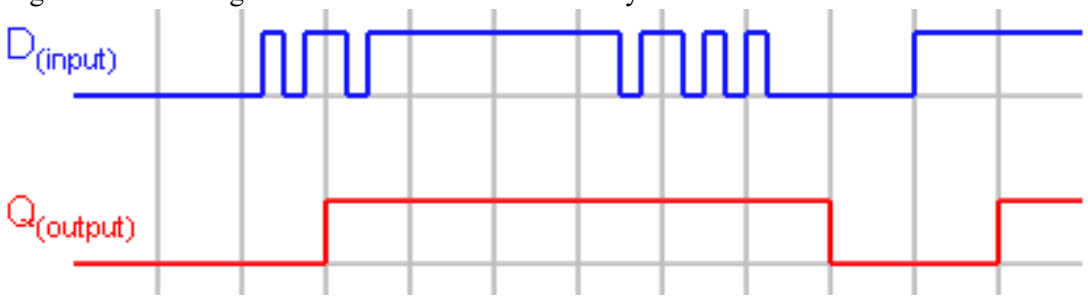



Figure 1. D Flip-Flop Inputs vs Outputs with Clock Rising Edge Every Second Line

## Procedure

The clock and counter circuit is made up of two smaller circuits: the clock signal, and the program counter. The clock signal contains three 555 timers (see Project 2.2) in different modes. The three modes are: *astable* mode, *monostable* mode, and *bistable* mode. The timers in astable and monostable mode serve as clock signals, while the timer in bistable mode is simply used to switch between the other 2.

Both timers constantly output a clock signal, but only one of the two is selected, and passed on to the rest of the circuit (see Figure 1).

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 3 | NE555P Timer |
| 15 | Fixed Resistor |
| 3 | 0.1 µF Capacitor |
| 3 | Momentary Push Button |
| 2 | SN74LS00N Quad NAND Gate |
| 1 | SN74LS161 Counter IC |
| 1 | SPDT Slide Switch |
| 7 | Square LED |
| 1 | 100 KΩ Trim Potentiometer |
| ~ | Wires |

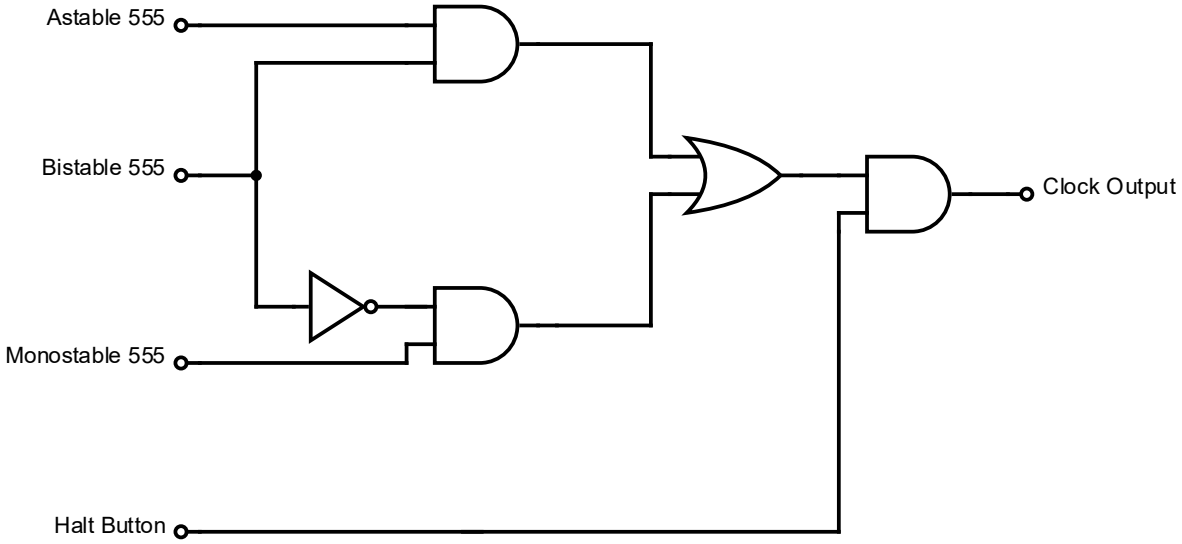



Figure 1. Logical Connections to Combine Clock Signals

In addition to the three inputs from the 555 timers, there is a halt button. This effectively disables the clock output. Working backwards from the clock output, the output can only ever be high when both inputs are also high. The halt button is wired in a pullup configuration, and normally high. This means that when the button is pressed, the clock output will remain low.

A 555 timer in bistable mode has two stable states: high and low; its current state is controlled by a slide switch. Referring to Figure 1, when the bistable 555 timer, or selector input, is high, one input in the top AND gate is high, and one input in the bottom AND gate is low as it passes through an inverter gate. This means that in this state, no matter what the state of the monostable 555 is, its AND gate will always be low. Additionally, in this state, the top AND gate simply echoes the state of the astable 555. When the selector circuit is low, the opposite behavior happens. Finally, the two circuits are combined using OR logic. This allows the user to select either clock signal using a slide switch.

Astable mode (see Figure 3) refers to a process where there is no stable state; the output continuously and autonomously switches between high and low states. It generates a square wave.

The 555 contains a series of voltage dividers to provide reference voltages of 1.67 V and 3.33 V (see Figure 2). When the circuit is first switched on, the reset comparator (see Figure 3) which is comparing the reference voltage of 1.6 V on the non-inverting input, to a voltage of 0 V on the inverting input, is high. Since 1.6 V is greater than 0 V, the set comparator will output a high signal, resetting the SR Latch. This low signal is stored until the set pin is high. Since the LED is connected to the inverted output, it is on in this state.

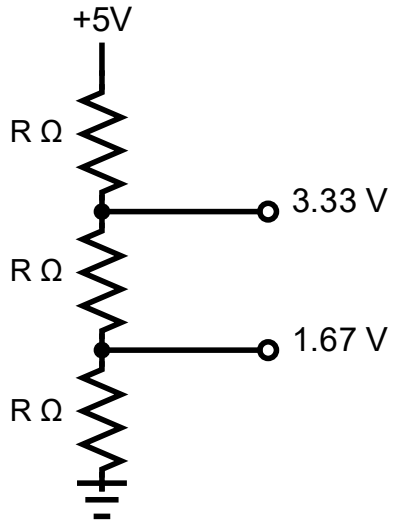



Figure 2. Voltage Dividers in the 555 Timer

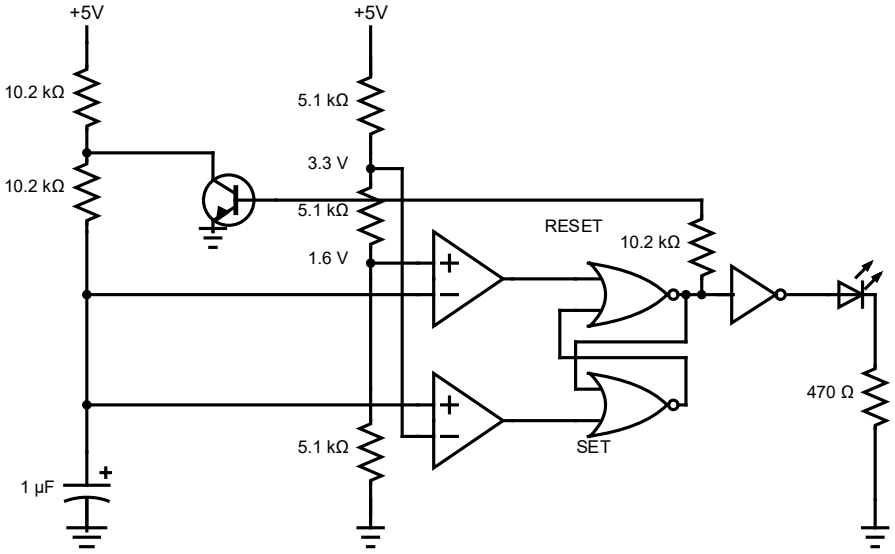



Figure 3. The 555 Time Machine in Astable Mode

At the same time, the capacitor pictured in Figure 3 begins to charge. When it eventually charges, beyond the reference voltage of 3.3 V, the set comparator outputs a high signal, since the voltage of the capacitor presented on the non-inverting input of the bottom op amp is greater than the reference voltage of 3.3 V presented on the inverting input. This causes the LED to switch off as its state is inverted. Additionally, the NPN transistor connected to the capacitor through a resistor switches on, which discharges the capacitor slowly. Once it discharges below 1.67 V, the SR latch is reset, and the LED turns back on. At this point, the circuit is back in its original state, and repeats the process.

The frequency at which this process occurs is dependent on the value of the two resistors in the top left of Figure 3, as well as the capacitor. By keeping the value of the capacitor constant, and changing the value of one resistor, the frequency can be changed. The clock circuit of chump utilizes a potentiometer in place of a resistor, allowing the user to modify the clock between a wide range of frequencies.

Monostable mode (see Figure 4) refers to a process where there is one stable state. It remains in the low (stable) state until an input is received, at which point the output goes high, for a set amount of time, which based off the configuration of a resistor-capacitor pair. In the context of the CHUMP clock, a 555 timer in monostable mode is used to debounce push button, allowing for a manual clock signal that does not occasionally produce undesired clock cycles.

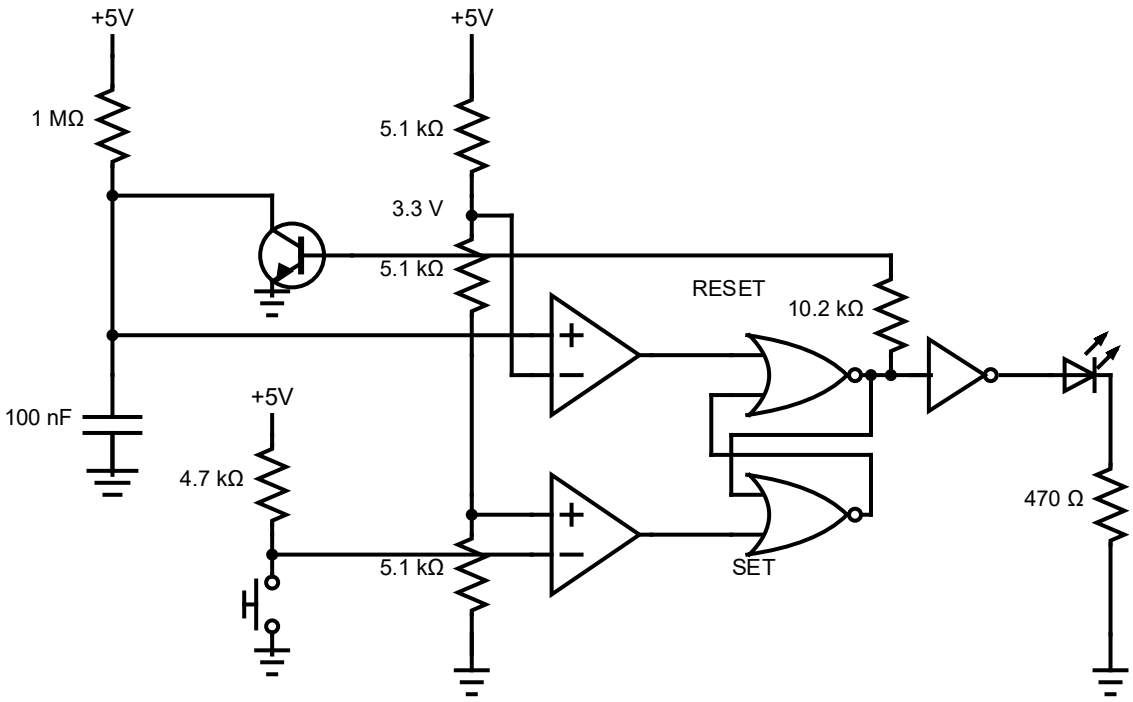



Figure 4. The 555 Time Machine in Monostable Mode

In a similar fashion to astable mode, monostable mode makes use of the 555 timer's reference voltages (see Figure 2). When the pushbutton is not active, since it is wired in a pullup configuration, the set comparator compares the reference voltage of 1.67 V to 5 V. Since 1.67 V is not greater than 5 V, it is low.

**Figure 5. SR Latch Truth Table**

| S | R | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0 | Latched | Latched |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | Undefined | Undefined |

Additionally, the reset comparator compares 0 V on the capacitor to the reference voltage of 3.3 V. Since 0 V is not greater than 3.3 V, the reset comparator is also low. This causes the NOR gate to remain high (see Figure 5), which discharges the capacitor through the transistor. In this configuration the circuit is stable; it will not change without additional input. Once the button is pressed, the set comparator goes high, as 0 V is less than the reference voltage of 1.67 V. This both causes the LED to go on, and the discharge capacitor to disable, putting it into its unstable state. As a result, the capacitor begins to charge. Once it reaches a charge greater than 3.3 V, the reset comparator goes high, which resets the SR latch (see Figure 5), puts it back into its stable state with the LED off. During the time that the LED is on, the circuit will ignore any additional input, or bounces of the pushbutton.

Each clock signal is then fed into logic gates, as described above (see Figure 1), and the final clock output goes into the SN74LS161 IC, which is a synchronous counter (see Figure 6).

The SN74LS161 is used as the program counter for the rest of the CHUMP build. As a synchronous counter, it makes use of a clock signal input on pin 2 (see Figure 6). When properly conditioned, the chip counts in binary on pins 11-14, with the count increasing on the rising edge of the clock signal. Since it is a 4-bit counter, it counts from 0-15.

When the load pin, which is normally high, is pulled low, the counter jumps to the number presented on pins on the next rising edge of the clock signal, simulating a branching instruction. It remains at that count until the load pin returns high, at which point it starts counting up normally.

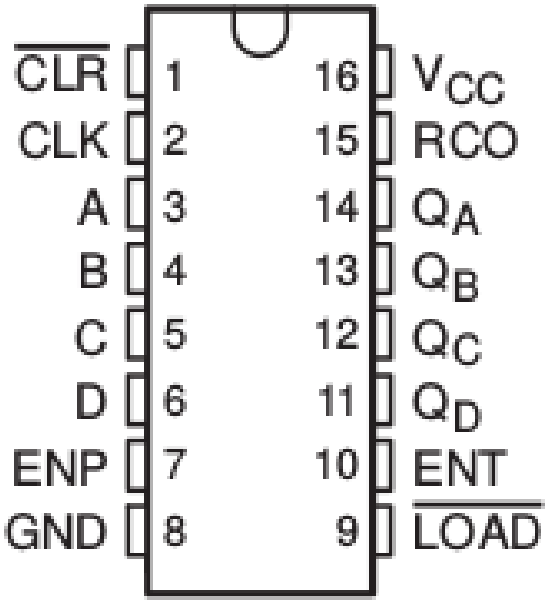



Figure 6. SN74LS161 Pinout

For counting, the SN74LS161 makes use of 4 D flip-flops (see Figure 7). A D flip-flop, which is built upon an SR flip-flop, has one input, D, as well as a clock input.

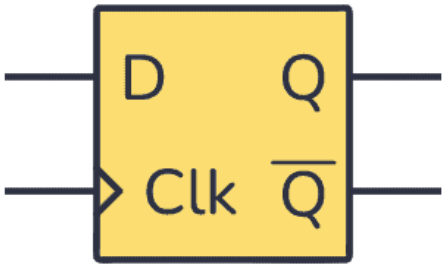



Figure 7. D Flip-flop

Unlike an SR latch and flip-flop, which can be set and reset, the D flip-flop has 1 input, which is echoed to Q on the rising edge of the clock signal.

Since it is a flip-flop, not a latch, the output can only change on the rising edge of the clock pulse. A D flip-flop is built using 2 AND gates, a NOT gate, and an SR latch (see Figure 9). By wiring the $\overline{Q}$ output to the D input, the Q output will automatically toggle on each rising clock cycle.

**Figure 8. D Flip-Flop Table**

| CLK | D | $Q_{Current}$ | $Q_{Next}$ |
|---|---|---|---|
| ↑ | 0 | 0 | 0 |
| ↑ | 0 | 1 | 0 |
| ↑ | 1 | 0 | 1 |
| ↑ | 1 | 1 | 1 |

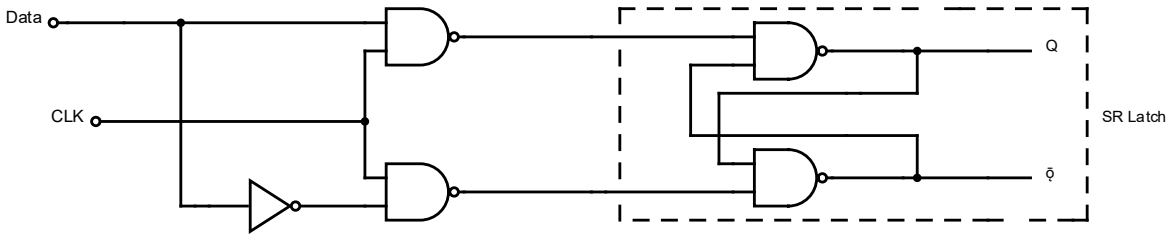



Figure 9. D Flip-Flop Made from SR Latch

While properly conditioned to count, the program counter makes use of the toggle feature of the D flip-flop. Each of the 4 outputs has its own D flip-flop (see Figure 10). In this configuration, the $\overline{Q}$ output of the least significant output ($Q_A$) is effectively wired to its own input. This means that $Q_A$ toggles its state at each rising edge of the clock signal.

The D flip-flop attached to $Q_B$ is conditioned using the many logic gates to be in the latched state (Q connected to D) until $Q_A$ transitions from 0-1. At this point, $\overline{Q}$ is passed to D. This means that $Q_B$ toggles on every falling edge of $Q_A$. Each subsequent output ($Q_C$ and $Q_D$) is conditioned using additional logic gates to toggle on the falling edge of the previous output (see Figure 10).

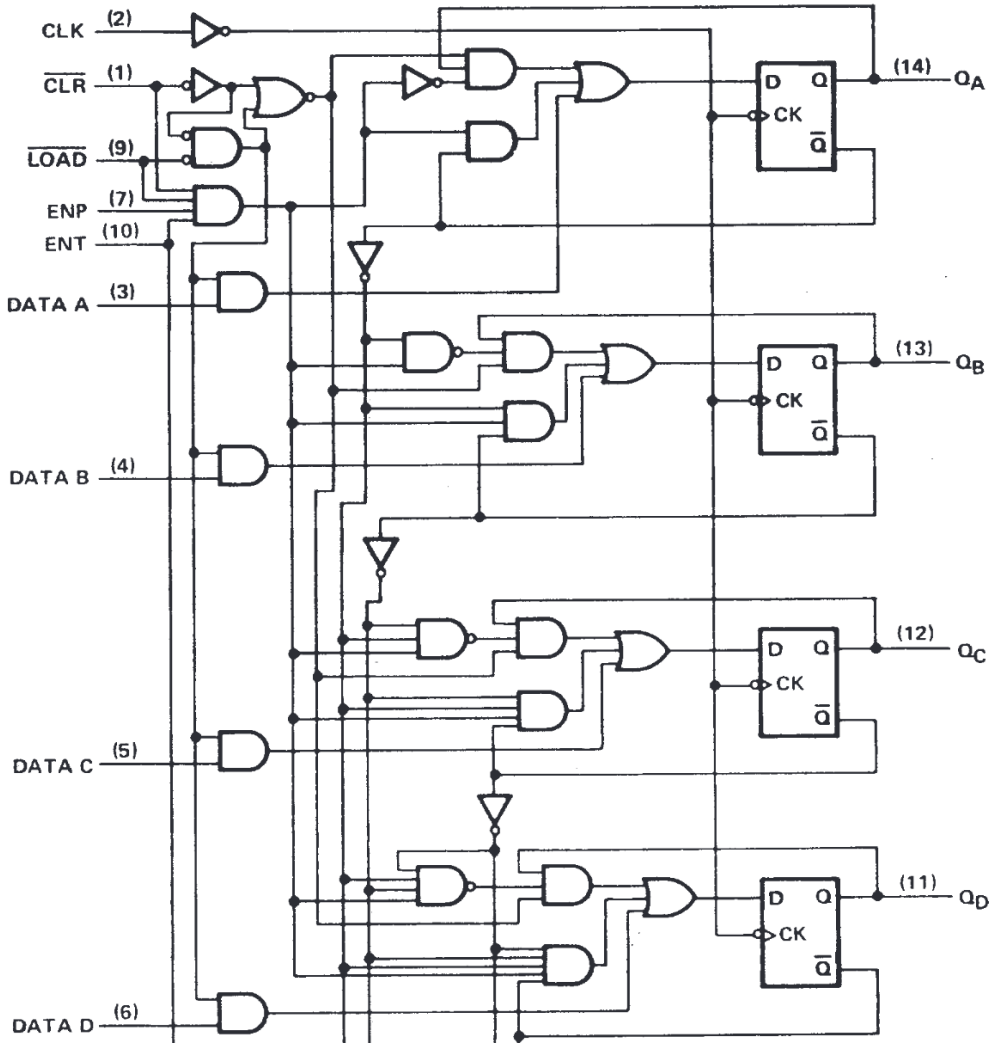


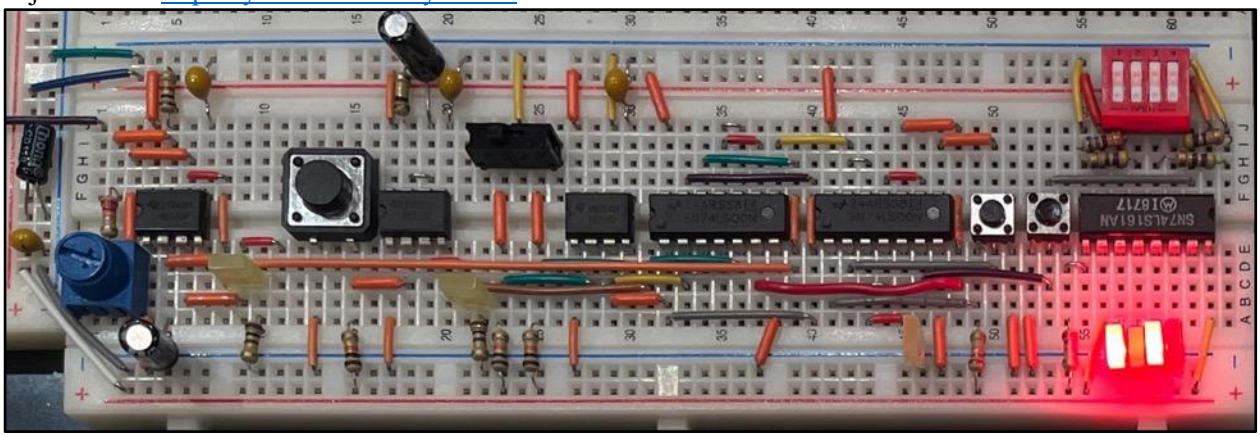

Figure 10. Program Counter IC Internals

Another way to look at a binary counter, such as the program counter, is as a frequency division circuit. The LSB flashes at a rate of half the clock; the next-most significant bit flashes at a quarter the clock frequency; the next flashes at an eighth, and so on. The result is a simulated binary counter.

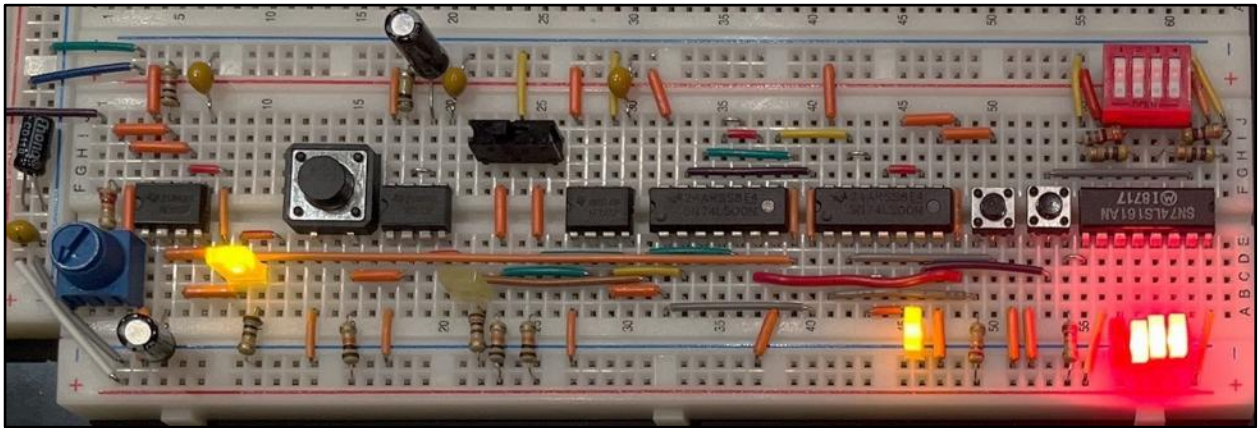
The output of the program counter is fed to the rest of CHUMP through a 4-bit wide bus. The count it provides is used by CHUMP to determine which address of the program EEPROM to read from.

## Media

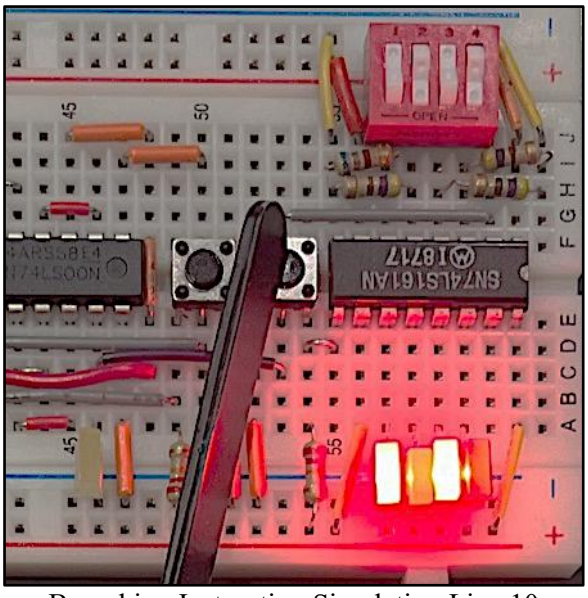
Project video: https://youtu.be/Lt8fisyeRXM



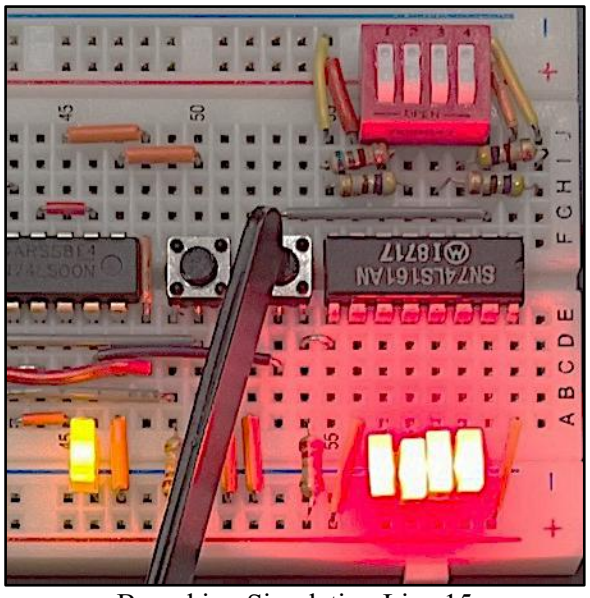
Program Count of 10, Falling Edge



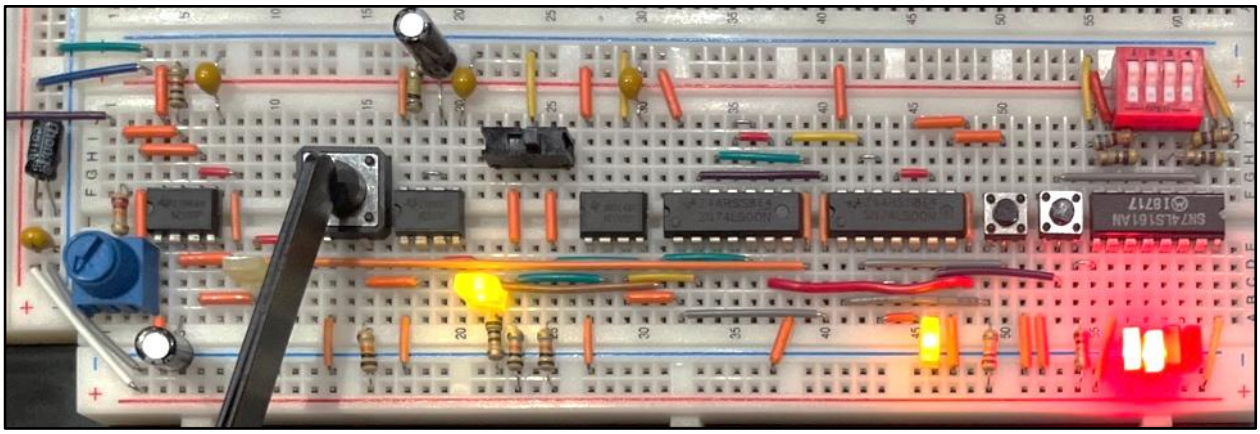
Program Count of 7, Rising Edge in Astable Mode



Branching Instruction Simulation Line 10



Branching Simulation Line 15



Program Count of 12, Rising Edge in Monostable Mode

## Reflection

I don't have a whole lot to say other than: CHUMP is going to be a great project. I really enjoyed learning about how this thing works, and upon studying Feinberg's sketch a bunch, I think I am beginning to truly understand how it works (from a high-level perspective). Learning about the program counter was a little bit confusing, only because there are multiple chips with very similar names on the same datasheet. I ended up writing a bunch of my report about the SN74161 (and JK flip-flops) instead of the SN74LS161 (which uses D flip-flops), before realizing that I had used the wrong schematic.

For the most part, time management was pretty good during this project; I had my report written by the Thursday, then did my video Monday night, so I had time to think about how I wanted to present it. Overall, it was a great start to CHUMP, and I can't wait to get cracking on the next stage.

## Project 3.2.2 Program EEPROM

### Purpose

The purpose of the program *EEPROM* (Electronically Erasable Programmable Read Only Memory) is to feed the rest of CHUMP (specifically the control ROM) with the op code and constant for each given input from the program counter.

### Reference

Project description:
http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#PROGRAMEEPROM

### Theory

There are several different types of ROMs (Read Only Memory). The most basic of which is the standard ROM, which can only be read, and is factory-programmed. The next type is the PROM (Programmable ROM), which starts off blank, and can be programmed a single time. When the PROM can be written to multiple times, it is called an EPROM (Erasable PROM). An EPROM can be written to many times, and is erased using ultraviolet (UV) light. In this configuration, when the die of the chip is exposed to UV light, the contents of the chip are erased. Finally, there is the EEPROM (Electronically EPROM), which can be erased completely electronically.

Combinational logic serves an important purpose in any circuit, whether simple or complex. By combining simple logic gates, any desired truth table can be created.

For example, the commonly used CD4511 seven segment display decoder chip (see Figure 1) consists of combinational logic. While it is a simple way to produce a desired truth table, more complex, and irregular truth tables require the use of many logic gates.

When flexibility is required, an EEPROM can be utilized to replace combinational logic. An EEPROM offers a major advantage: it can be reprogrammed at any time. In applications where flexibility is needed, such as a programmable computer, this is a major benefit.

In the context of CHUMP, multiple EEPROMs are employed to act as decoders, in a similar fashion as the CD4511 was used in the Counting Circuit (see Project 1.4). In this configuration, the EEPROMs can be used as hexadecimal decoders for seven segment displays, mapping each instruction to an op code, and storing the program to be executed.

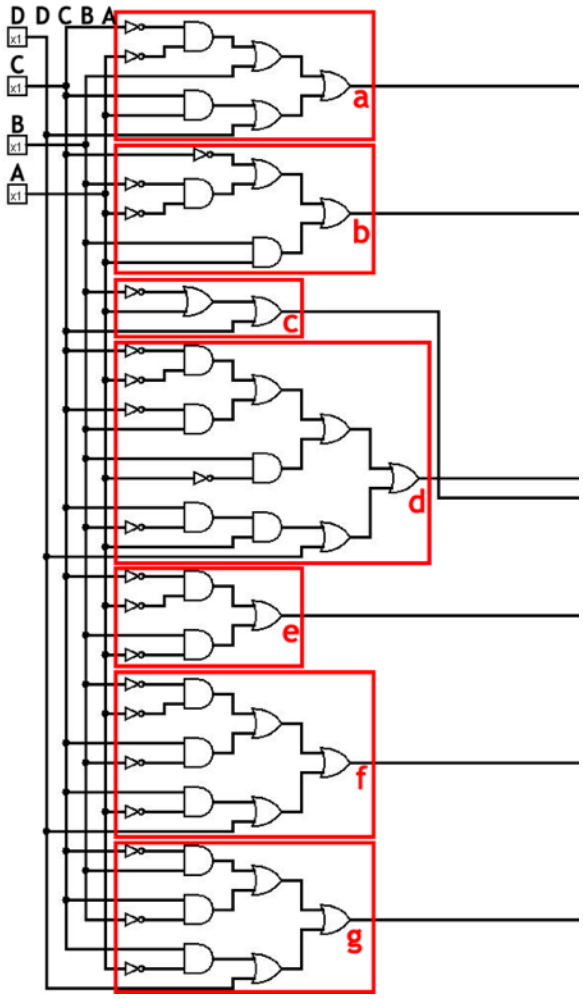



Figure 1. CD4511 Schematic

## Procedure

The program EEPROM is made from a blank EEPROM. It is programmed using an Arduino Nano, and 2 shift registers. In this configuration, the Arduino selects the address to set using the shift registers, then writes data stored in an array to the selected address using its digital pins. Since the Arduino digital pins can be used as inputs or outputs, the programmer can also be used to read the EEPROM.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 4 | Microchip 28C16A EEPROM |
| 1 | Dual Digit 7-Segment Display |
| 1 | Arduino Nano |
| 2 | SN74HC595N Shift Register |
| 1 | Clock and Counter Circuit |
| ~ | Wires |

To program the EEPROM, the following sequence is used: firstly, the desired 11-bit address is selected on pins A0-A10 (see Figure 1). Next, the output enable pin is set high. Since it is an active low, to disable the output (and allow it to be programmed) it must be pulled high. Then, the information to be stored to the selected address is presented on the I/O pins. Finally, a low pulse is sent to the [active low] write enable pin.

The pulse sent to the write enable pin has specific parameters; the pulse width must be between 100 and 1000 nanoseconds. Once the pulse is sent, the EEPROM will have stored the state of the 8 I/O pins.

An EEPROM is a form of non-volatile storage; unlike RAM, which is a type of volatile memory, it does not require power to retain its information. After being disconnected from power, the information will continue to be retained when powered up again

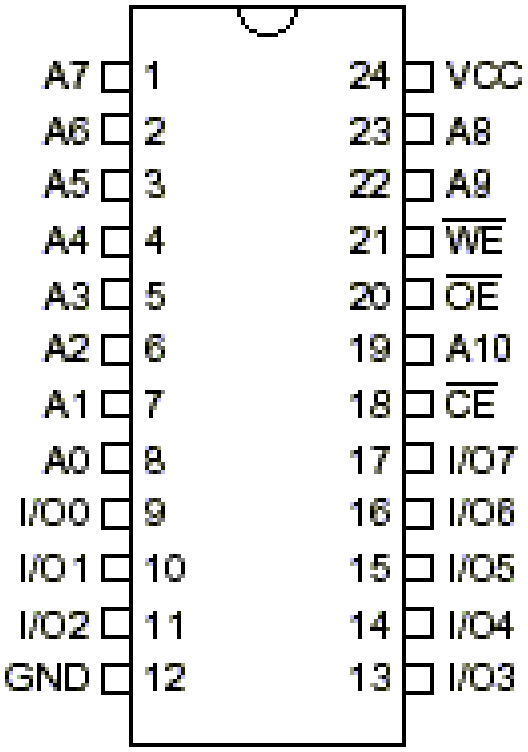



Figure 1. 28C16A EEPROM Pinout

The Arduino programmer contains 2 basic functions: `readEEPROM(uint16_t address)`, as well as `writeEEPROM(uint16_t address, uint8_t data)`. The former simply shifts out the address provided, then reads the states of the I/O pins, and allows the user to view the contents of the EEPROM at any valid address. The latter shifts out the address provided, presents the value of data on the I/O pins, then pulses the write enable pin low. This allows the user to set any valid address with any desired 8-bit value.

One EEPROM contains 2048 bytes, while one CHUMPanese program consumes only 16 bytes. This means that a single EEPROM could contain 128 CHUMPanese programs. To do this, a technique called *paging* is utilized. Paging divides the EEPROM into sections of a certain size, in this case 16, allowing for 128 pages, or programs.

The program ROM is used to store the hexadecimal control codes for CHUMP. Each control code, consists of a single byte. In this configuration, the high nibble is known as the *OpCode*, while the low nibble is known as the *Constant* (see Figure 2). The OpCode is the 4-bit code that tells CHUMP what to do; it is the action, or instruction. The constant is that data that the instruction is to be performed on. For example, if the control code aimed to LOAD 2, LOAD would be represented by the OpCode, while the 2 would be stored in the constant

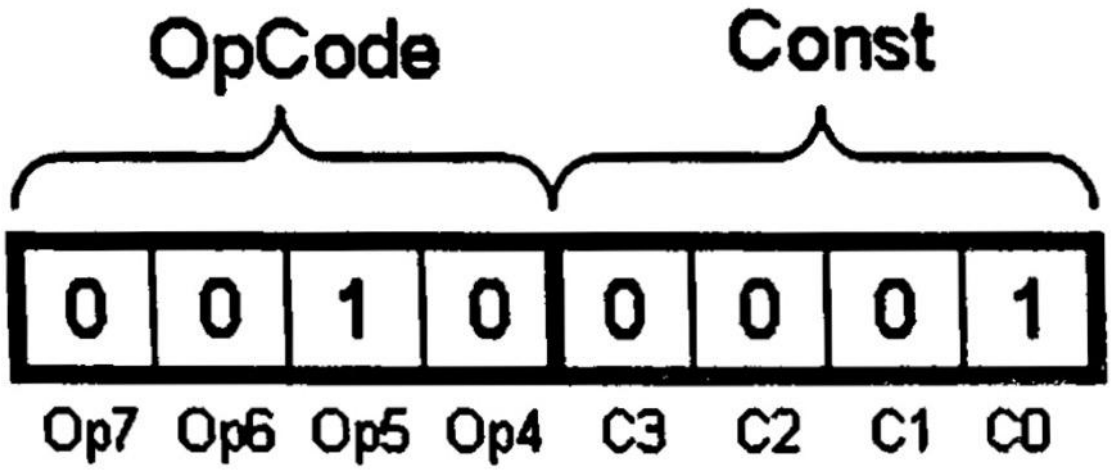



Figure 2. Control Code Components

In the wider context of CHUMP, the OpCode is the code that is fed into CHUMP's control ROM (see Figure 3). The control ROM is used to replace combination logic (see Theory section).

The control ROM receives the OpCode, and is programmed in advance to correctly interface with CHUMP based on the instruction. For example, for the instruction LOAD, the control ROM would correctly set the control lines to load the constant provided by the program ROM into the ALU (see Figure 3).

Since the OpCode is 4-bits, there are a total of 16 possible instructions. In the default CHUMP configuration, 14 instructions are populated, or 2 groups of 7 instructions. Each instruction has constant version, as well as a memory version.

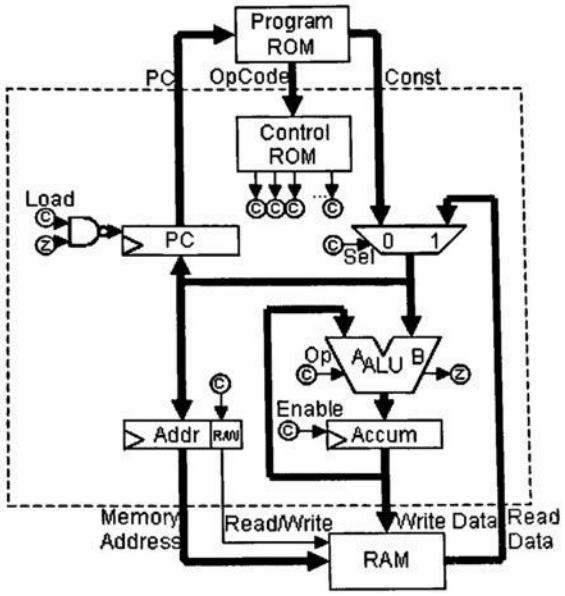



Figure 3. CHUMP Schematic

Having a constant and memory version of each instruction allows CHUMP to work with the value stored in memory, or the value in the lower nibble of the control code (the constant). These two different versions of the same instruction are interpreted as separate CHUMPanese instructions.

In this stage of CHUMP, the control ROM is not present; instead, the control code is displayed on 2 seven-segment displays in hexadecimal, using two EEPROMs as hexadecimal decoders. In this configuration, each EEPROM decodes a nibble into a hexadecimal digit.

The EEPROM is flashed with a segment map (see Figure 5), which maps each address (input) to a combination of segments. To avoid ambiguity with numbers, the hexadecimal characters A, C, E, and F are displayed as uppercase, while b and d are displayed as lowercase (see Figure 5).

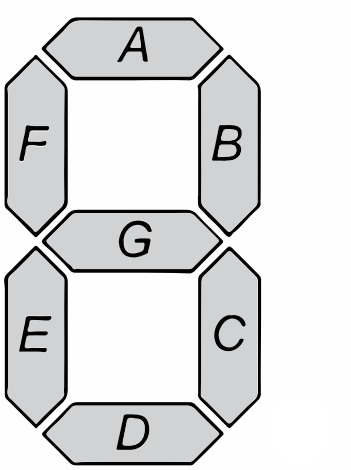



Figure 4. Seven-Segment Display Layout

| Figure 4. EEPROM Seven Segment Display Decoder Map | | | | |
|---|---|---|---|---|
| **Digit** | **Byte (Hex)** | **Byte (Binary)** | **Segments (see Figure 4)** | **Displayed Character** |
| 0 | 0x3F | 0011 1111 | A, B, C, D, E, F | 0 |
| 1 | 0x06 | 0000 0110 | B, C | 1 |
| 2 | 0x5B | 0101 1011 | A, B, D, E, G | 2 |
| 3 | 0x4F | 0100 1111 | A, B, C, D, G | 3 |
| 4 | 0x66 | 0110 0110 | B, C, F, G | 4 |
| 5 | 0x6D | 0110 1101 | A, C, D, F, G | 5 |
| 6 | 0x7D | 0111 1101 | A, C, D, E, F, G | 6 |
| 7 | 0x07 | 0000 0111 | A, B, C | 7 |
| 8 | 0x7F | 0111 1111 | A, B, C, D, E, F, G | 8 |
| 9 | 0x6F | 0110 1111 | A, B, C, D, F, G | 9 |
| A | 0x77 | 0111 0111 | A, B, C, E, F, G | A |
| B | 0x7C | 0111 1100 | C, D, E, F, G | b |
| C | 0x39 | 0011 1001 | A, D, E, F | C |
| D | 0x5E | 0101 1110 | B, C, D, E, G | d |
| E | 0x79 | 0111 1001 | A, D, E, F, G | E |
| F | 0x71 | 0111 0001 | A, E, F, G | F |

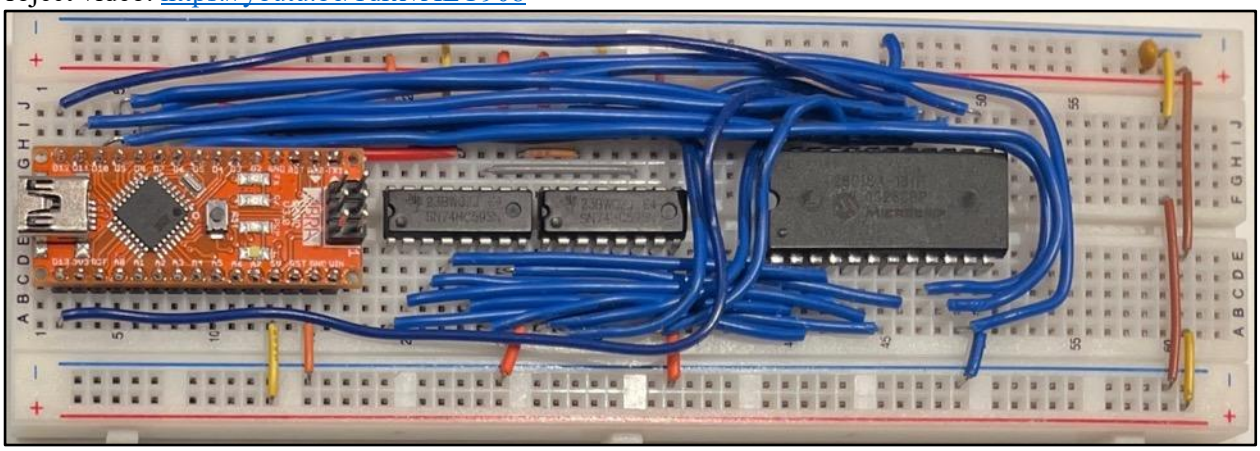
Code

```
// PROJECT    : EEPROM Burner
// AUTHOR     : R. Jamal (adapted from Ben Eater)
// PURPOSE    : To burn an EEPROM with data from an array
// COURSE     : ICS4U-E
// DATE       : 26 10 2024
// MCU        : MEGA328P (Nano)
// STATUS     : Working
#define EEPROM_D0 5          //Lowest EEPROM I/O pin
#define EEPROM_D7 12         //Highest EEPROM I/O pin
#define WRITE_EN PB5         //Write enable pin
#define LATCH PD3            //Shift register latch pin
#define DATA PD2             //Shift register serial pin
#define CLOCK PD4            //Shift register clock pin

byte data[] = {0x82, 0x10, 0x21, 0x62, 0xa0, 0xff, 0xff, 0xff,  //Program codes
               0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
               0x05, 0x60, 0x07, 0x61, 0x80, 0x10, 0x81, 0x51,
               0xaa, 0xcd, 0x00, 0x62, 0xc0, 0x01, 0x62, 0xc1
              };

void setup() {
  Serial.begin(57600);                                        //Begin serial comm
  DDRD |= 1 << DATA | 1 << CLOCK | 1 << LATCH;                //Set SR pins as outputs
  DDRB |= 1 << WRITE_EN;                                      //Set write enable as out
  PORTB |= 1 << WRITE_EN;                                     //Pull write enable high

  Serial.print("Programming EEPROM");                         //Signify start of process
  for (uint16_t address = 0; address < sizeof(data); address++)  //Loop through
    writeEEPROM(address, data[address]);                      //Write each address
  printContents();                                            //Print EEPROM contents
}
void loop() {}                                                //Nothing to do

void setAddress(int address, bool outputEnable) {             //Sets address on SR
  PORTD &= ~(1 << LATCH);                                     //Pull latch low
  ShiftPortD(DATA, CLOCK, MSBFIRST, (address >> 8) | (outputEnable ? 0x00 : 0x80));
  ShiftPortD(DATA, CLOCK, MSBFIRST, address);                 //Shift out second byte
  PORTD |= 1 << LATCH;                                        //Pull latch high
}
void ShiftPortD(uint8_t dataPin, uint8_t clockPin, uint8_t bitOrder, uint8_t val) {
  for (uint8_t curBit = 0; curBit < 8; curBit++)  {
    if (!bitOrder) {                                          //LSBFIRST = 0
      if (val & (1 << curBit)) PORTD |= (1 << dataPin);       //Set dataPin HIGH
      else PORTD &= ~(1 << dataPin);                          //Set dataPin LOW
    }
    else {                                                    //MSBFIRST = 1
      if (val & (1 << (7 - curBit))) PORTD |= (1 << dataPin);  //Set dataPin HIGH
      else PORTD &= ~(1 << dataPin);                          //Set dataPin LOW
    }
    PORTD |= (1 << clockPin);                                 //Set clockPin HIGH
    PORTD &= ~(1 << clockPin);                                //Set clockPin LOW
  }
}

byte readEEPROM(int address) {                                //Reads EEPROM at address
  DDRD &= ~(0b11100000);                                      //Set I/O pins as inputs
  DDRB &= ~(0b00011111);                                      //Set I/O pins as inputs
  setAddress(address, true);                                  //Set address on SR
  byte data = 0;                                              //Buffer for read data
  for (int pin = EEPROM_D7; pin >= EEPROM_D0; pin -= 1)       //Loop through bits
    data = (data << 1) + digitalRead(pin);                    //Fetch each bit
  return data;
```

```
}

void writeEEPROM(uint16_t address, byte data) {           //Write data to EEPROM
  setAddress(address, false);                             //Set address on SR
  DDRD |= 0b11100000;                                     //Set I/O pins as outputs
  DDRB |= 0b00011111;                                     //Set I/O pins as outputs
  for (int pin = EEPROM_D0; pin <= EEPROM_D7; pin += 1) { //Loop through
    digitalWrite(pin, data & 1);                          //Set bit of current pin
    data = data >> 1;                                     //Move to next bit
  }
  PORTB &= ~(1 << WRITE_EN);                              //Pull write enable low
  delayMicroseconds(1);                                   //Wait 1 microsecond
  PORTB |= 1 << WRITE_EN;                                 //Pull write enable high
  delay(10);                                              //Wait 10 ms
}

void printContents() {                                    //Prints EEPROM contents
  Serial.println("");                                     //Goes to next line
  for (int base = 0; base <= 255; base += 16) {           //Loops through
    byte data[16];                                        //Creates buffer for data
    for (int offset = 0; offset <= 15; offset += 1) {     //Loops through
      data[offset] = readEEPROM(base + offset);           //Downloads data
    }
    char buf[80];                                         //Second buffer
    sprintf(buf,                                          //Parses data
"%03x:%02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x
%02x",
    base, data[0], data[1], data[2], data[3], data[4], data[5],
    data[6], data[7], data[8], data[9], data[10], data[11],
    data[12], data[13], data[14], data[15]);

    Serial.println(buf);                                  //Prints buffer
  }
}
```

## Media

Project video: https://youtu.be/1drxvALU908



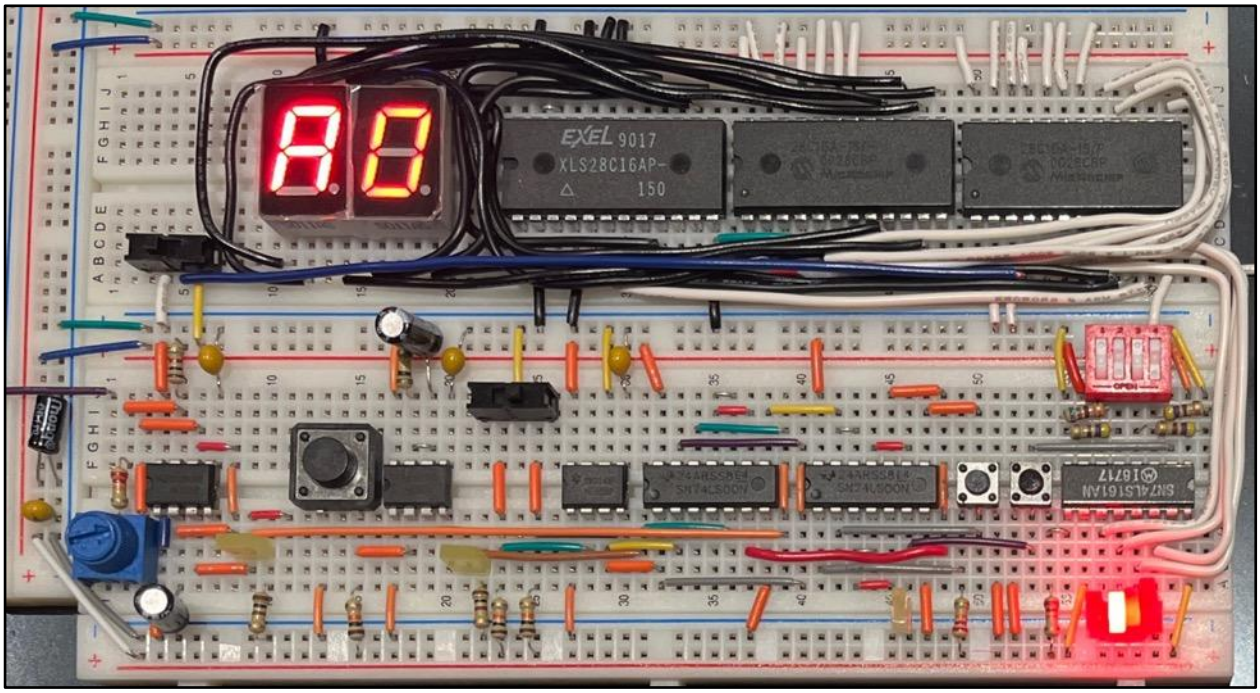
EEPROM Programmer

```
000:82 10 21 62 a0 ff ff ff ff ff ff ff ff ff ff ff
010:05 60 07 61 80 10 81 51 aa cd 00 62 c0 01 62 c1
020:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
030:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
040:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
050:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
060:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
070:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
080:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
090:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0a0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0b0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0c0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0d0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0e0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0f0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
```

Serial Monitor Reading Program EEPROM

```
000:3f 06 5b 4f 66 6d 7d 07 7f 6f 77 7c 39 5e 79 71
010:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
020:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
030:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
040:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
050:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
060:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
070:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
080:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
090:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0a0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0b0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0c0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0d0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0e0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0f0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
```

Serial Monitor Reading Decoder EEPROM




Control Code A0 (GOTO 0)

## Reflection

For me, this is the stage that CHUMP all comes together. At this point, I (think) that I mostly understand how CHUMP works. The explanation of the OpCode and constant in the CHUMP workbook paved the way for me.

As far as actually building this project went, it was somewhat of a nightmare. There were three or four issues that I spent at least 2 hours debugging, only for the solution to be something relatively trivial. For example, forgetting that I commented out a `pinMode()` statement, or forgetting to wire the second shift register's latch pin, and rebuilding the whole circuit. Unfortunate or not, as I've said many times before, that's engineering, and that's how you learn.

## Project 3.2.3 Arithmetic and Logic Unit (ALU)

### Purpose

The purpose of the Arithmetic and Logic Unit (ALU) is to perform the mathematical operations that CHUMP demands of it using combinational logic.

### Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#ALU

ALU datasheet: https://www.ti.com/lit/ds/symlink/sn54s181.pdf?ts=1730913668778

https://medium.com/@werowe/how-computers-do-math-and-logic-5e1467b55bc6

### Theory

There are 2 basic functions that nearly any computer performs: adding numbers (arithmetic) and comparing 2 numbers (logic). The ability to add numbers allows the computer to perform any of the four fundamental arithmetic operations. Subtraction comes from adding a positive and negative number: $a - b = a + (-b)$. Multiplication comes from adding a number a certain number of times: $a \times b = a + a + a \ldots + a$, where the operation is executed $b$ times. Finally, division comes from checking the number of times one number can be subtracted from another: $\frac{a}{b} = a - b - b \ldots - b$, where the quotient is given by the number of times that the operation is executed until a result of 0 is obtained.

The second function, logic, involves the comparison of two values. A computer is able to compare numbers using predetermined logic functions. These include the binary operators AND and OR, as well as the unary operator NOT. Logic functions do not follow the same algebraic rules as traditional arithmetic functions.

To distribute a logic function across a set of brackets, rather than traditional algebra, *DeMorgan's Theorem* (see Figure 1) is utilized. DeMorgan's Theorem has two parts: the first states that distributing a NOT function across two terms being ORed results in the inverse of each term being ANDed: $(A + B)' = A'B'$. The second part states that distributing a NOT function across two terms being ANDed results in the inverse of each term being ORed: $(AB)' = A' + B'$. DeMorgan's Theorem can be thought of as an extension to the universality of gates (see Project 1.4B), allowing the creation of new combinational logic truth tables with a limited set of existing gates.


Figure 1. DeMorgan's Theorem

## Procedure

The SN74LS181 ALU is responsible for the arithmetic and logic within CHUMP. The rest of CHUMP is structured around the ALU, adhering to its requirements. The previous stages of CHUMP (see Project 3.2.1, 3.2.2) serve to feed the ALU the correct data to produce the desired outputs.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | SN74LS181 ALU IC |
| 4 | 1 KΩ Bussed Resistor Network |
| 17 | Square LEDs |
| ~ | Wires |

As a 4-bit ALU, the SN74LS181 (see Figure 1) has a total of 16 functions in logic mode and an additional 16 functions in arithmetic mode.

When M (see Figure 1) is low, S0-S3 selects an arithmetic function; when M is high, S0-S3 selects a logic function (see Figure 2).

To decode the instruction from the program ROM, the control ROM is utilized. When the opcode from the program ROM (see Project 3.2.2) is fed into the control ROM, it outputs the corresponding ALU instruction on the S and M pins. The instruction is then executed using either the data provided in the lower nibble of the instruction code, or some data stored in RAM.

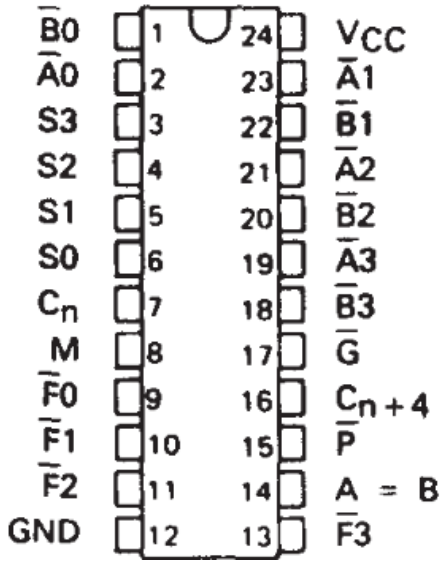


Figure 1. SN74LS181 ALU Pinout

| Figure 2. ALU Functions Overview | | |
|---|---|---|
| **Selection (S0-S3)** | **Logic Function** | **Arithmetic (No Carry)** |
| 0 | $F = \bar{A}$ | $F = A$ |
| 1 | $F = \overline{A + B}$ | $F = A + B$ |
| 2 | $F = \bar{A}B$ | $F = A + \bar{B}$ |
| 3 | $F = 0$ | $F =$ MINUS 1 (2's Comp) |
| 4 | $F = \overline{AB}$ | $F = A$ PLUS $A\bar{B}$ |
| 5 | $F = \bar{B}$ | $F = (A + B)$ PLUS $A\bar{\bar{B}}$ |
| 6 | $F = A \oplus B$ | $F = A$ MINUS $B$ MINUS 1 |
| 7 | $F = A\bar{B}$ | $F = A\bar{B}$ MINUS 1 |
| 8 | $F = \bar{A} + B$ | $F = A$ PLUS $AB$ |
| 9 | $F = \overline{A \oplus B}$ | $F = A$ PLUS $B$ |
| 10 | $F = B$ | $F = (A + \bar{B})$ PLUS $AB$ |
| 11 | $F = AB$ | $F = AB$ MINUS 1 |
| 12 | $F = 1$ | $F = A$ PLUS A |
| 13 | $F = A + \bar{B}$ | $F = (A + B)$ PLUS $A$ |
| 14 | $F = A + B$ | $F = (A + \bar{B})$ PLUS $A$ |
| 15 | $F = A$ | $F = A$ MINUS 1 |

Within the ALU, there are distinct circuits for each function (see Figure 3). The select pins (S0-S3) and M pin control a 5-bit multiplexer that selects the correct arithmetic or logic circuit. Each green block in Figure 3 depicts a combinational logic circuit.

When all output pins are high, the A=B flag goes high. In the wider context of CHUMP, this feature of the ALU is used for the `IFZERO` instruction. By setting one of the ALU's inputs to 0, setting the other input to the value of the accumulator, comparing the two and monitoring the A=B pin, CHUMP can determine whether the current value stored in the accumulator is 0 or not. By modifying the control ROM bits, this capability can be extended to testing for any 4-bit value.

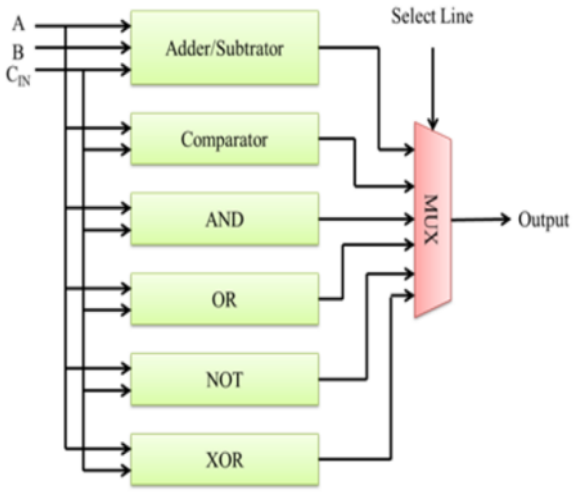



Figure 3. ALU Overview

To perform subtraction, CHUMP does a form of addition involving the negative of one term (see Theory section). To make a number negative, CHUMP makes use of the *Two's Complement* of the number (see Figure 4). To find the Two's Complement, of a number, the number is subtracted from 0.

For the positive numbers, the positive of the number is used (see Figure 4). In the case of the negative numbers, the subtraction must be carried out. When performed by hand, the following shortcut can be used: each bit is inverted, then the result is added to one, which yields the results in Figure 4.

**Figure 4. 4-Bit Two's Complement Numbers**

| Decimal | Two's Complement | Unsigned Binary Equivalent |
|---|---|---|
| -8 | 1000 | 8 |
| -7 | 1001 | 9 |
| -6 | 1010 | 10 |
| -5 | 1011 | 11 |
| -4 | 1100 | 12 |
| -3 | 1101 | 13 |
| -2 | 1110 | 14 |
| -1 | 1111 | 15 |
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |

After finding the *Two's Complement* of $B$ when performing $F = A + (-B)$, regular addition is carried out. For example, when $A = 5$ and $B = 6$, the result is: $F = 5 + (-6) = -1$.

Using the unsigned binary equivalent values from Figure 4, $A = 5$ and $B = 10$ when simplifying for $-B$. In this configuration, $F = 5 + 10 = 15$. Referring to Figure 4, the equivalent value of 15 in a 4-bit signed environment is -1, which is equivalent to the answer obtained using traditional arithmetic.

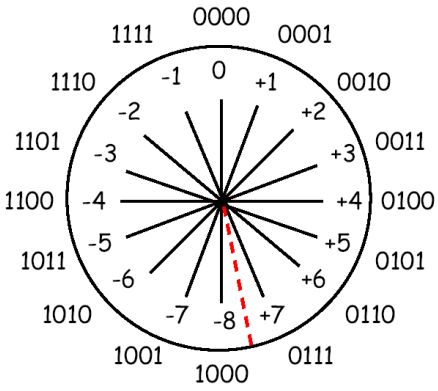


Figure 5. Two's Complement Circle

To showcase the individual functions of the SN74LS181, an ALU explorer circuit is built. In this configuration, two separate 4-bit DIP rocker switch are wired to the A and B inputs in a normally low configuration, with their states echoed to two separate 4-bit *AT bargraphs*. An AT bargraph (designed by Atticus Tiplady ACES '25) consists of square LEDs in a 3D-printed bargraph enclosure (see Figure 6). Pins F0-F3 are displayed on an additional AT bargraph, and the A=B pin is connected to a single indicator LED.

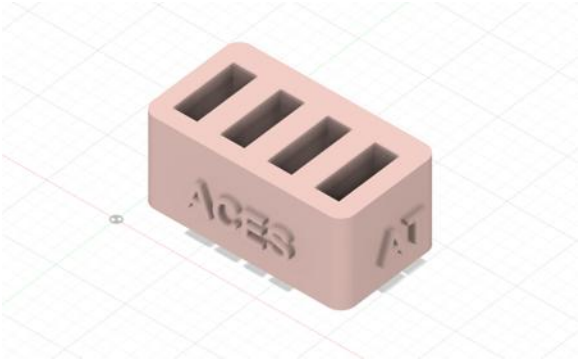
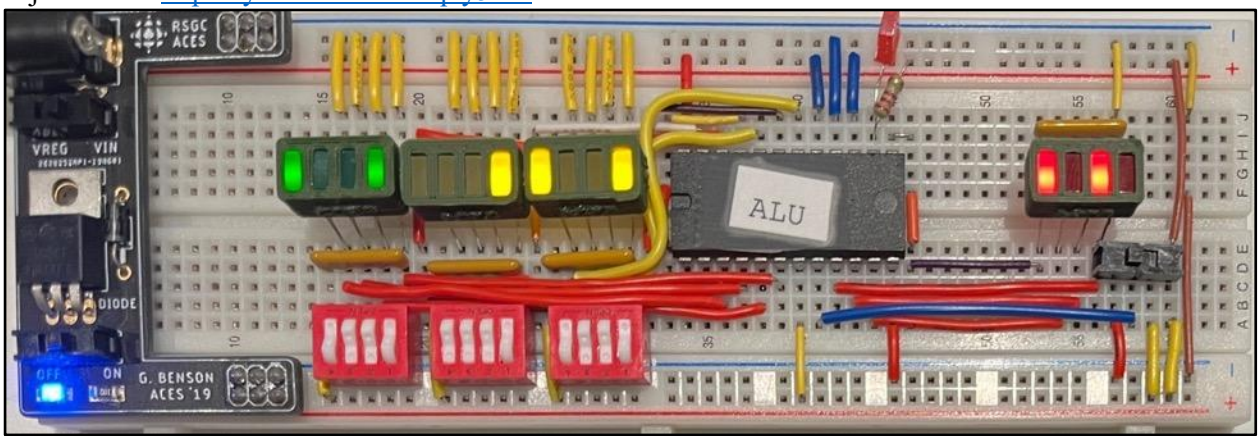


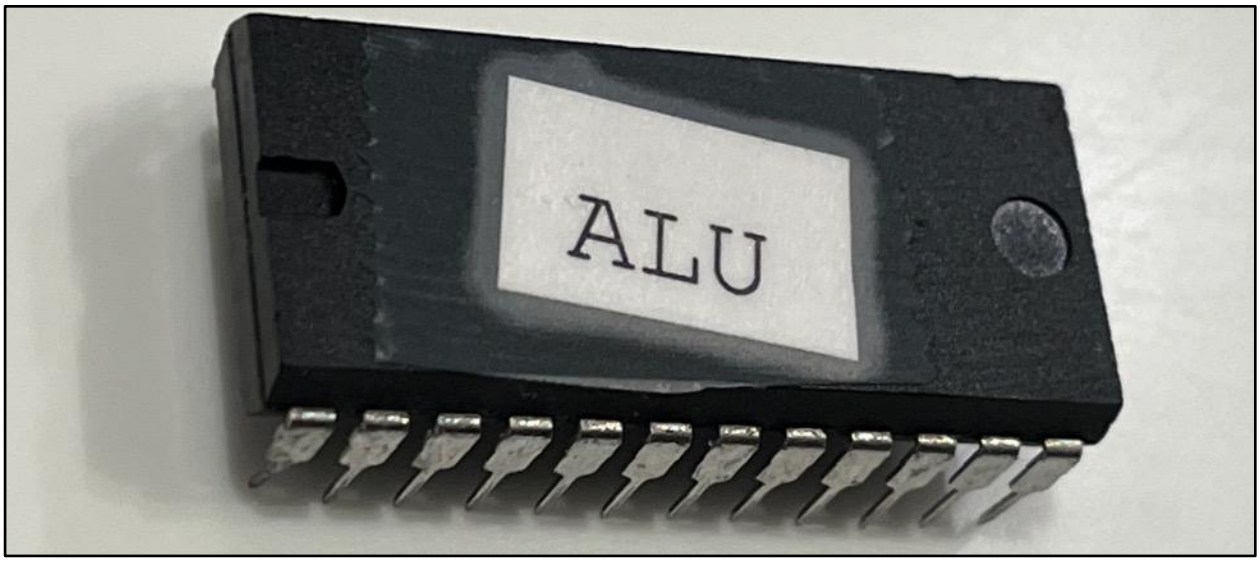
Figure 6. AT Bargraph

## Media

Project video: https://youtu.be/OOHltquy5dM

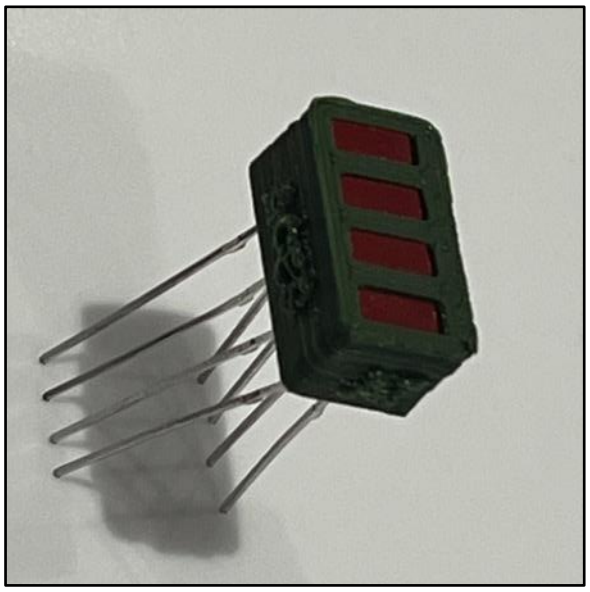


ALU Explorer

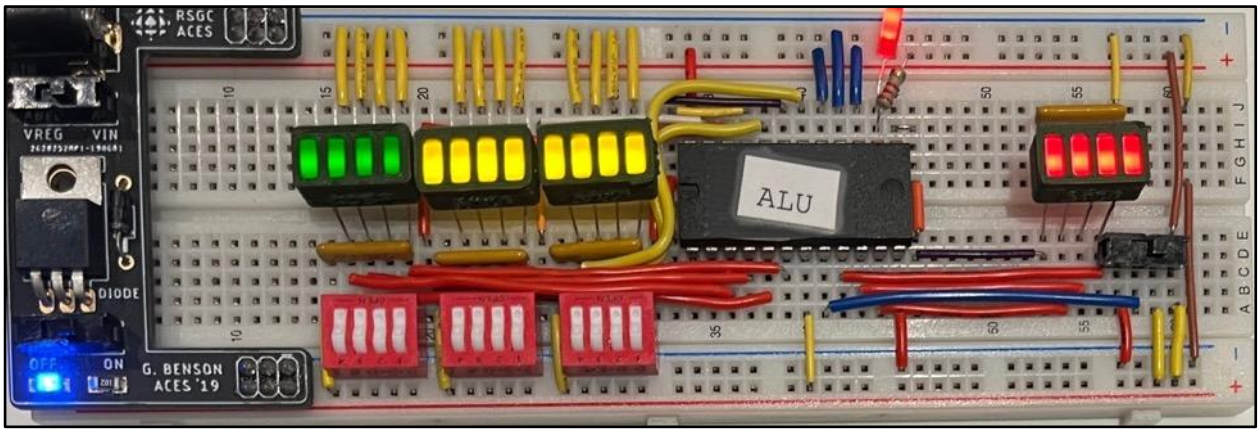



SN74LS181

AT Bargraph (Unpopulated)



AT Bargraph (Populated)




ALU Explorer All Bits High

## Reflection

This has been a pretty hectic project. Similar to the last one, I was out of town for the weekend, but unlike the last one, I was not able to complete it before leaving. I ended up bringing my build with me, and working on it whenever I was in the car.

I think that this segment of the project has let me realize the potential of CHUMP. A few days ago, I heard Mr. D'Arcy mention the possibility of using CHUMP as a microcontroller replacement. I think that this enhancement would really take CHUMP from a theoretical project to a truly practical project. Also, I think that if I was able to add another instruction that could slow down the clock signal to create a delay, or switch to a manual pulse to wait for user input it could really increase the functionality of CHUMP.

Maybe I'm walking on rainbows here, but by the end of CHUMP, I hope that I am able to implement it as a microcontroller, add some sort of instruction that alters the clock to either wait for user interaction or just a delay, and depending on the difficulty, extend the number of lines to 256. I know that it would be tough to turn CHUMP into an 8-bit computer, but I think I have a somewhat fleshed-out plan for creating what I like to call a hybrid 8-bit computer.

My modified CHUMP would basically include 8-bit and 4-bit parts. Since I just want to extend the number of lines to 256, the ALU could stay 4-bits, I could chain on another program counter, and there is still lots of space on the program ROM. I also do not take any issue with the 16-instruction limitation, so the control bus could stay 4-bits wide. The only major issue I foresee with my approach is that it would not be possible to jump to an instruction beyond line 15 as the RAM and constant number are both only 4-bits. My only solution would be to use 2 clock ticks and 2 separate instructions for a branch; `GOTOLOW` would set the first program counter, and `GOTOHIGH` would set the second program counter. Regardless, that is a problem for a future stage of CHUMP.

In the meantime, it's time to continue the development of my ISP.

## Project 3.2.5 CHUMP: Final Build

### Purpose

The final stage of CHUMP is a fully programmable large breadboard circuit that can add and compare numbers. Using these features, the inclusion of an I/O register allows CHUMP to be used in a microcontroller-like fashion. Additionally, a single CHUMPanese program can contain a maximum of 256 instructions and CHUMP can address a maximum of 16 locations in RAM.

### Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#Build

### Theory

One of the most important ways to measure a processor's capabilities and limitations is its *bitness*. The bitness of a processor refers to the number of bits that can be processed in a single clock cycle. A processor of *n* bits can only perform operations involving numbers that are *n* bits long, which is a direct limit of the ALU. To process numbers greater than *n*, multiple clock cycles are required, performing the task serially.

In addition to the limitations imposed by the ALU, a CPU's bitness also usually defines the width of the processor's busses. Due to an *n*-bit wide busses, the program counter, control ROM and address buffers can only address *n*-bit wide busses. This effectively limits an *n*-bit processor to having a maximum of $2^n$ distinct instructions, lines of code, and locations in memory (see Figure 1). Many of these limitations can be overcome by taking multiple clock cycles per instruction.

**Figure 1. Bitness Limitations**

| $n$ | $2^n$ |
|---|---|
| 1 | 1 |
| 2 | 4 |
| 4 | 16 |
| 8 | 256 |
| 16 | 65536 |
| 32 | 4294967296 |
| 64 | $1.84467441 \times 10^{19}$ |

An instruction is a number that corresponds to a series of pre-defined control codes that setup the components in the computer for the desired function. For example, an instruction that writes to RAM would require the RAM to be in write mode. This means that the control bit pertaining to RAM would have to be high (for an active high write) or low (for an active low write).

There are two main types of processors: *reduced instruction set computing* (RISC) and *complex instruction set computing* (CISC). A RISC processor (such as CHUMP) has fewer, more basic instructions. A CISC processor has many complex instructions that perform a higher-level feature than a RISC instruction. For example, to perform addition to a value stored in RAM, a RISC processor would have to load the value, perform the addition, and store the new value back to RAM, which would take 3 full clock cycles. A CISC processor would have a specific instruction designed to add numbers in RAM, and would perform the addition in fewer clock cycles.

Both RISC and CISC offer advantages over the other; the use of either architecture is highly dependent on the application of the processor. An advantage of the RISC architecture is its low power consumption.

## Procedure

The final version of CHUMP is a 4-bit computer that can add, subtract, and AND numbers. The ALU has many additional functions (see Project 3.2.3), however as a result of the overhead of the basic instructions (see Figure 1), there are not enough OpCodes available to utilize each function.

The only instruction that is conditional is the `IFZERO` instruction, which checks if the value stored in the accumulator is 0. Using the add and subtract function, this instruction can be used to test for any value in the accumulator.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 2 | 74LS161 Program Counter |
| 1 | 74LS174 Flip-Flops |
| 2 | 74LS477 Accumulator |
| 1 | 74LS189 RAM |
| 1 | 74LS181 ALU |
| 2 | Microchip 28C16A EEPROM |
| 2 | 74LS157 2 to 1 MUX |
| ~ | LEDs |
| ~ | Wires |

**Figure 1. CHUMP Instructions and Functions**

| Instruction | OpCode | Function |
|---|---|---|
| LOAD const | 0000 | accum ← const |
| LOAD IT | 0001 | accum ← [ADDR] |
| ADD const | 0010 | accum ← accum + const |
| ADD IT | 0011 | accum ← accum + [IT] |
| SUBTRACT const | 0100 | accum ← accum - const |
| SUBTRACT IT | 0101 | accum ← accum – [IT] |
| STORETO const | 0110 | [const] ← [const] |
| STORETO IT | 0111 | [IT] ← accum |
| READ const | 1000 | addr ← const |
| STORETOPORT const | 1001 | [const] ← output reg ← accum |
| AND const | 1010 | accum ← accum AND const |
| LOADPORT | 1011 | accum ← input reg |
| GOTO const | 1100 | pc ← const |
| GOTO IT | 1101 | pc ← [IT] |
| IFZERO const | 1110 | if(!accum) pc ← const |
| OR const | 1111 | accum ← accum OR const |

The inclusion of the `AND const` instruction (OpCode 1010) allows CHUMP to test whether an individual bit is active or not. This is an important instruction for the I/O feature of CHUMP (see Purpose section) as a bit represents a specific pin. By ANDing the value of the input register with a mask, and subtracting the value of the desired bit, CHUMP can decide whether a specific input pin is high or low. It can then proceed to the appropriate line with an if/else statement. In CHUMPanese, an if/else statement can be written using a combination of `GOTO` and `IFZERO` instructions. In this configuration, the operand of the `IFZERO` instruction points to the line to be executed when the accumulator is zero (`if` case). Similarly, the code in the case when the accumulator is not zero (`else` case) follows the `IFZERO`, and a `GOTO` is used at the end to bypass the `if` case.

Conceptually, the standard CHUMP consists of several 4-bit busses and a 10-bit control bus that connects the various components together. The program counter starts at 0 and increments on each rising edge of the clock. The outputs of the program counter are fed directly into the program ROM (see Figure 2) which contains an OpCode and operand for each value of the program counter. The OpCode corresponds to a series of control bits and is decoded by the control ROM (see Figure 2).

A multiplexer selects between a RAM operand and the operand provided in the low nibble of the program ROM. Before any instruction involving an operand stored in RAM can be executed, an address must be put on the address buffer, which stores the RAM address for use on the next clock cycle.

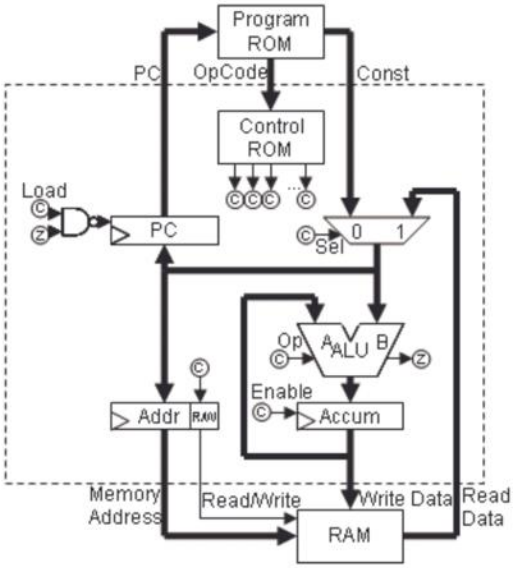



Figure 2. Standard CHUMP Schematic

Once the operand goes through the multiplexer, it is automatically stored on the address buffer, and can be used by the B input of the ALU, or the program counter depending on the instruction. The control ROM is programmed with the contents of Figure 3, relating the arbitrary OpCode to a useable set of control codes for CHUMP.

**Figure 3. CHUMP OpCodes and Control Codes**

| OpCode | MUX | ALU | Mode | Carry | Accum | RAM | PC |
|---|---|---|---|---|---|---|---|
| 0000 | 0 | 1010 | 1 | X | 0 | 1 | 0 |
| 0001 | 1 | 1010 | 1 | X | 0 | 1 | 0 |
| 0010 | 0 | 1001 | 0 | 1 | 0 | 1 | 0 |
| 0011 | 1 | 1001 | 0 | 1 | 0 | 1 | 0 |
| 0100 | 0 | 0110 | 0 | 0 | 0 | 1 | 0 |
| 0101 | 1 | 0110 | 0 | 0 | 0 | 1 | 0 |
| 0110 | 0 | 1010 | 1 | X | 1 | 0 | 0 |
| 0111 | 1 | 1010 | 1 | X | 1 | 0 | 0 |
| 1000 | 0 | 1010 | 1 | X | 1 | 1 | 0 |
| 1001 | 1 | 1010 | 1 | X | 1 | 0 | 0 |
| 1010 | 0 | 1011 | 1 | X | 0 | 1 | 0 |
| 1011 | 1 | 1010 | 1 | X | 0 | 1 | 0 |
| 1100 | 0 | 1100 | 1 | X | 1 | 1 | 1 |
| 1101 | 1 | 1100 | 1 | X | 1 | 1 | 1 |
| 1110 | 0 | 0000 | 1 | X | 1 | 1 | 1 |
| 1111 | 1 | 1110 | 1 | X | 1 | 1 | 0 |

CHUMP has 3 locations that it can store values: the RAM and accumulator, which can hold data indefinitely, and the address buffer, which can hold data for a single clock cycle. The accumulator is meant to temporarily store values and can hold exactly one 4-bit value which is pulled directly from the output of the ALU. On any instruction that manipulates numbers, the accumulator is enabled, which means that it takes the value from the ALU on the rising edge of the clock cycle. The RAM stores values in the long term and has 16 different 4-bit locations. Each location is can be accessed with a 4-bit address value.

Despite its 4-bit nature, CHUMP has an 8-bit program counter which allows a sequence of up to 256 instructions to be executed in a single program. In this configuration, two program counters are daisy-chained together and wired directly to the program ROM. In order to properly achieve 8-bit branching on a 4-bit bus, 2 clock cycles are required. Figure 4 depicts the inputs of the second program counter connected directly to the address buffer. This allows a branch instruction to be executed by placing the first 4 bits on the address bus (using `READ const`), and then executing a standard `GOTO` with the second 4 bits. The clock signal first passes through a flip-flop in order for the branch to occur at the correct time. The inputs of the flip-flop are fed by a *diode OR gate* with inputs from the first program counter and control ROM.

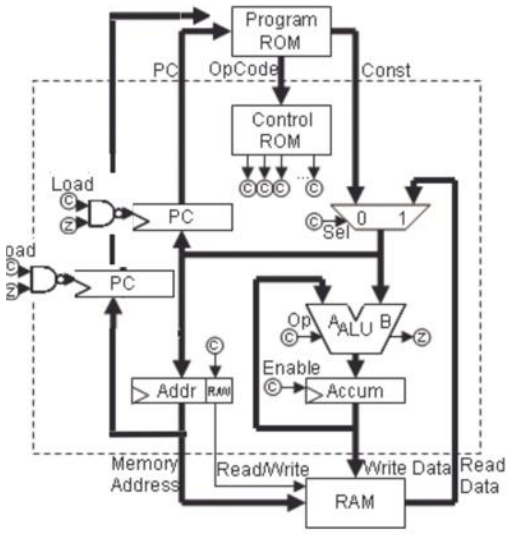



Figure 4. Dual Program Counter CHUMP

A diode OR gate provides the same function as a traditional OR gate, but is built using diodes instead of transistors (see Figure 5). A diode is a component that only conducts when the voltage on the anode exceeds the voltage on the cathode. In Figure 5, since the cathodes are pulled down, when both A and B are low, the diodes do not conduct, and the output is low. When either A or B is high, its respective diode conducts, providing a path to power which causes the output to be high.

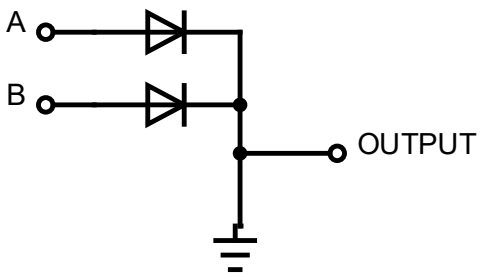



Figure 5. Diode OR Gate

Due to the double-usage of the address bus to store the high nibble of a branching instruction as well as the current address in RAM, `GOTO IT` instructions are limited to 4-bit values; a `READ` instruction must precede a memory instruction, but a `READ` instruction also must precede a branching instruction which creates a conflict for a memory branching instruction. To solve this, an additional control bit is used to clear the second program counter (set to 0) during a memory branching instruction. Finally, due to the limited use of `READ IT` and `IFZERO IT`, they have been eliminated to include alternative ALU instructions.

The final stage of CHUMP contains input and output registers that allow for user interaction (see Figure 6). In this configuration, an additional control ROM is utilized as several new control bits are required: 1 bit for the second multiplexer that selects between RAM and the input register, 1 bit to act as a clock for the output register's clock source, and 1 bit to reset the second program counter during a memory branching instruction. Finally, 1 new control bit is necessary to control the first MUX due to the optimization of important instructions. When combined with the ability to run a program containing 256 instruction codes, the addition of I/O allows CHUMP to perform relatively advanced microcontroller operations. CHUMP can perform digital comparison operations to set certain inputs based on set conditions.

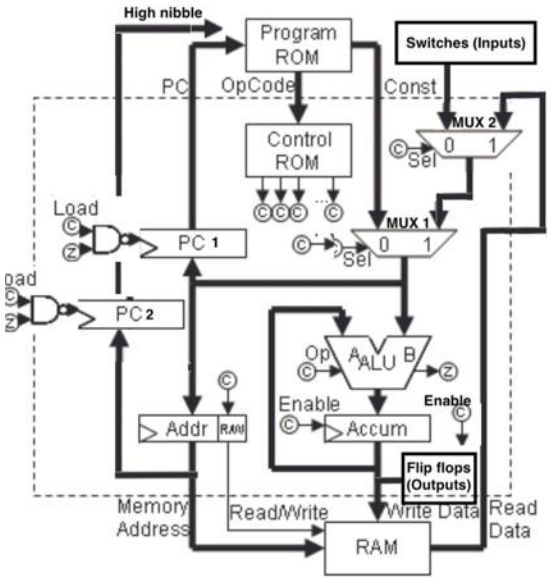



Figure 6. CHUMP I/O Registers

The input register of CHUMP adds a second multiplexer. During a constant-operand instruction, the state of the second multiplexer does not matter as it does not involve the constant inputs. The input of the second MUX is disabled when the first MUX selects 0.

The second MUX switches between RAM and the input register. The second MUX is always set to input B, the only exception to this is the `LOADPORT` instruction (see Figure 7) which loads the value on the input

The output register of CHUMP consists of 4 flip-flops (74LS174) with their inputs connected directly to the output of the accumulator. The clock input of all 4 flip-flops is provided by a control bit. The control bit is only high during the `STORETOPORT` instruction, which transfers the value in the accumulator to the output register. To keep track of the output port, the value is stored to RAM simultaneously. In this configuration, the provided operand is the address of RAM to store to (see Figure 8). Any desired bit can then be set or cleared by recalling the value from RAM and performing a bitwise OR or AND, respectively.

| Figure 7. LOADPORT All Control Bits | |
|---|---|
| **Control Bit** | **Value** |
| MUX 1 | 1 |
| ALU | 1010 |
| Mode | 1 |
| Carry | X |
| Accum | 0 |
| RAM | 1 |
| PC 1 | 0 |
| MUX 2 | X |
| Output Register | 0 |
| PC 2 Reset (Act Low) | 1 |

| Figure 8. STORETOPORT All Control Bits | |
|---|---|
| **Control Bit** | **Value** |
| MUX 1 | 0 |
| ALU | 1010 |
| Mode | 1 |
| Carry | X |
| Accum | 0 |
| RAM | 0 |
| PC 1 | 0 |
| MUX 2 | X |
| Output Register | 1 |
| PC 2 Reset (Act Low) | 1 |

One application of the I/O registers is the ability to generate PWM-like signals. By increasing the clock frequency of CHUMP, a PWM signal can be generated. Figure 9 depicts a CHUMPanese program that generates a PWM signal on the LSB of the output register, with its duty cycle dependant on the input register.

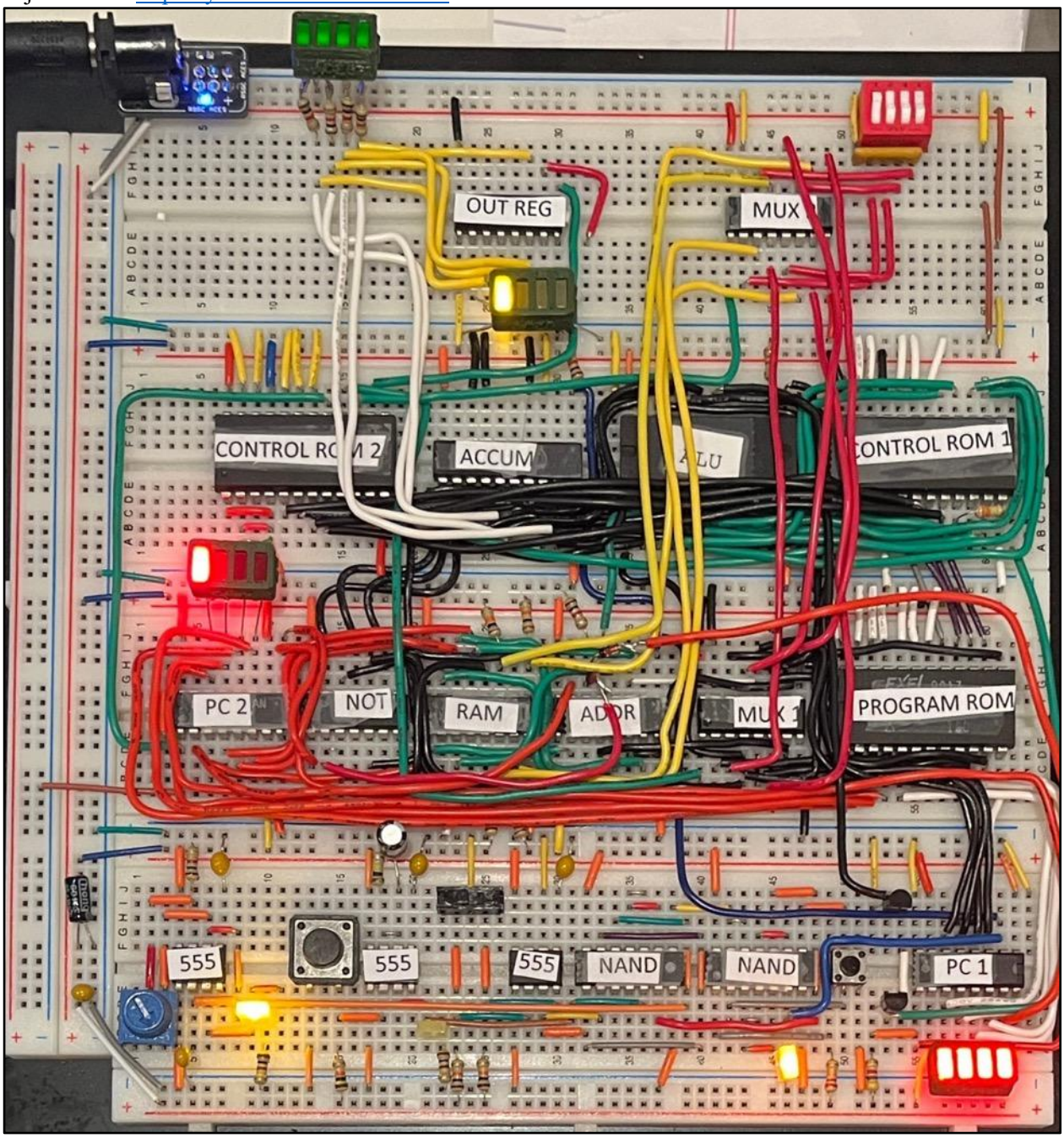
The program essentially counts, and on each increment of the count compares the index to the switch bank. When the sum of them overflows (is equal to 16), it turns on the PWM bit. Then, it waits for the count to get to 15 to turn off the pin and start over. This is an example of a 4-bit counter since it counts to 16.

**Figure 9. PWM CHUMPanese Program with Duty Cycle Based on Input Register**

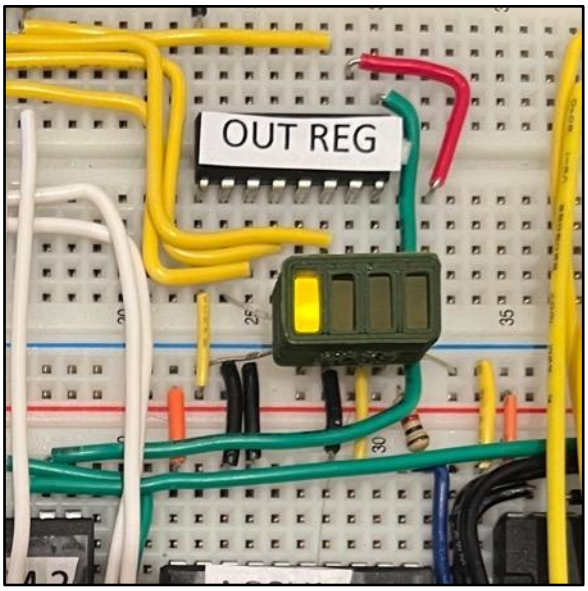
| Address | **High Level** | **Machine** | **CHUMPanese** | Comment |
|---|---|---|---|---|
| 0000 0000 | PORTB = 0 | 0000 0000 | LOAD 0 | accum ← 0 |
| 0000 0001 | ; | 1001 0000 | STORETOPORT 0 | [0] ← output reg ← accum |
| 0000 0010 | i = 0; | 0110 0001 | STORETO 1 | [1] ← accum |
| 0000 0011 | i++ | 1000 0001 | READ 1 | addr ← 1 |
| 0000 0100 | | 0001 0000 | LOAD IT | accum ← [1] |
| 0000 0101 | | 0010 0001 | ADD 1 | accum ← accum + 1 |
| 0000 0110 | ; | 0110 0001 | STORETO 1 | [1] ← accum |
| 0000 0111 | if(i+PINB==16){ | 1011 0000 | LOADPORT | input reg ← accum |
| 0000 1000 | accum = PINB + i | 1000 0001 | READ 1 | addr ← 1 |
| 0000 1001 | ; | 0011 0000 | ADD IT | accum ← accum + [1] |
| 0000 1010 | | 1000 0000 | READ 0 | addr ← 0 |
| 0000 1011 | } | 1110 1110 | IFZERO 14 | if(!accum) pc ←14 |
| 0000 1100 | else i++ | 1000 0000 | READ 0 | addr ← 0 |
| 0000 1101 | | 1100 0011 | GOTO 3 | pc ← 3 |
| 0000 1110 | | 1000 0000 | READ 0 | addr ← 0 |
| 0000 1111 | PORTB \|= 0b0001 | 1000 0000 | READ 0 | addr ← 0 |
| 0001 0000 | | 0001 0000 | LOAD IT | accum ← [1] |
| 0001 0001 | | 1111 0001 | OR 1 | accum ← accum OR 1 |
| 0001 0010 | ; | 1001 0000 | STORETOPORT 0 | [0] ← output reg ← accum |
| 0001 0011 | if(i==16) { | 1000 0001 | READ 1 | addr ← 1 |
| 0001 0100 | | 1000 0001 | READ 1 | addr ← 1 |
| 0001 0101 | | 0001 0000 | LOAD IT | accum ← [1] |
| 0001 0110 | | 1000 0000 | READ 0 | addr ← 0 |
| 0001 0111 | } | 1110 0001 | IFZERO 1 | if(!accum) pc ←1 |
| 0001 1000 | else i++ | 1000 0001 | ADD 1 | addr ← 1 |
| 0001 1001 | | 0110 0001 | STORETO 1 | [1] ← accum |
| 0001 1010 | | 1000 0001 | READ 1 | addr ← 1 |
| 0001 1011 | | 1100 0010 | GOTO 3 | pc ← 19 |

## Media

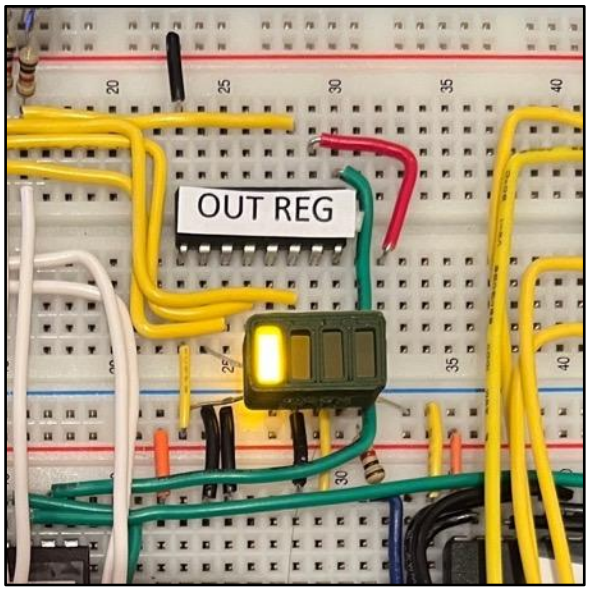
Project video: https://youtu.be/xfGtRVXh8Xc




CHUMP Full Build



Output Register, Minimum Duty Cycle



Output Register, Maximum Duty Cycle

Minimum Clock Frequency

Maximum Clock Frequency

## Reflection

This has been a phenomenal project. I can honestly say that I genuinely understand the entire CHUMP, and I fulfilled all of my wishes from the start of the project. Over the last month, many days were spent in the DES until 6 or 7 P.M. I was fascinated from the start with increasing the clock frequency from 1 Hz to a much higher frequency. I managed to get the frequency to 90.23 KHz, and in a way that my program did genuinely benefit from the additional processing power.

This project also answered one of my biggest questions about microcontrollers and computers (Arduinos in particular): last year, I remember asking Mr. D'Arcy how the Arduino IDE transfers the code you write onto the chip itself and how the processor is able to interpret the code. Now that I have written my own assembly code, converted it to machine language, and built the processor for it to run on, I think that I do understand how it works.

# Project 3.3 Long ISP: 8-Bit R/2R Audio Player

## Purpose

The purpose of this ISP, the 8-Bit Audio Player is to play rotary-encoder-selectable-by-name 8-bit, 16 KHz, mono wav files using an R/2R digital to analog converter (see Project 2.1) in place of a traditional I2C or SPI DAC.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/2425/ISPs.html#logs

Project proposal: http://darcy.rsgc.on.ca/ACES/TEI4M/2425/images/RJAudio.png

Project GitHub: https://github.com/rohan-development/3.3-Long-ISP-R2R-Audio-Player

PCB schematic: https://oshwlab.com/rjamal/ipod-thing

## Theory

There are 2 main classes of audio files: compressed audio, and uncompressed. The dominant audio file format in modern devices is MP3, which is a type of compressed audio. In order to play most standard audio files (regardless of compression), samples of the amplitude at a known sample rate must be obtained. In an uncompressed audio file, the raw sample values are already available in the file; in a compressed audio file, the samples must be obtained by applying an algorithm to the compressed data. These sample rates are then converted to an analog voltage, using a DAC, and amplified to be perceived as sound (see Figure 1).

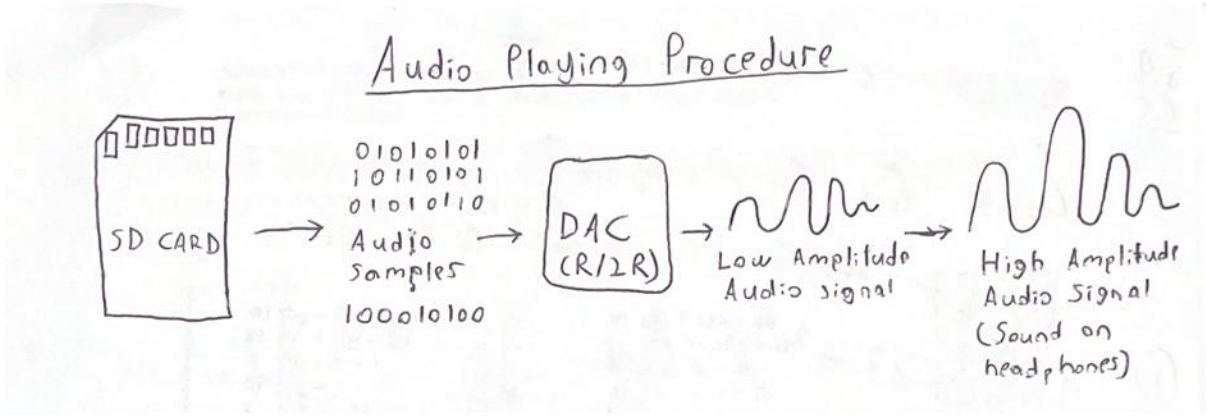



Figure 1. Audio Playing Sequence Using a DAC

When referencing the Atmega328P, most compression algorithms require significantly more computing power than available to playback in real time. For this reason, the R/2R audio player utilizes Waveform Audio File Format (WAV) files to store audio data, which simply store the raw discrete samples.

The basic concept of 8-Bit R/2R Audio Player comes from Project 2.1, the R/2R DAC. An R/2R ladder, or DAC, is a simple DAC that converts an 8-bit, discrete value, to an analog voltage between 0 and the supply voltage using the principle of voltage division. It included 8 switches that could be turned on or off, for a total of 256 steps between 0 and 5V. By connecting the 8 switch inputs to a microcontroller, a stored 8-bit digital value can be converted to an analog voltage. In this configuration, the R/2R DAC enables the microcontroller to emulate PWM with true analog voltage, instead of a duty cycle-based approximation.

## Procedure

On a very basic level, to play audio, the 8-Bit Audio Player makes use of an R/2R ladder and 8-bit, 16 KHz WAV files (see Theory section). In this configuration, an ATmega328P reads audio samples from an SD card, and sequentially outputs the samples onto the R/2R ladder at a rate of 16 KHz. To advance through the samples reliably at 16 KHz, a timer 1 interrupt is used. The raw signal from the output of the R/2R DAC is filtered through a capacitor, and then amplified through an LM386 audio amplifier, before finally being sent to headphones through a 3.5 mm audio jack.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 13 | Assorted Capacitors |
| 1 | 3.5 mm Audio Jack |
| 25 | ±0.1% 10 KΩ Resistor |
| 38 | Assorted Standard Resistor |
| 2 | Atmega328P SMD MCU |
| 1 | USB-C Connector |
| 1 | LM386 Audio Amplifier |
| 1 | MCP4551T Digipot |
| 2 | 16 MHz Crystal Oscillator |

The ATmega328P has only 2 KB of memory, while an average WAV file can be in excess of 5000 KB. To get around this limitation, a buffer is used. An 8-bit unsigned array with 256 locations (256 bytes) is created. In this configuration, the first 256 bytes are read from the SD card and placed in the buffer array. Then, a timer1-driven interrupt with a frequency of 16 KHz is initialized Each time the interrupt is triggered, an index is incremented. In the `loop()` function, the R/2R DAC is constantly updated with the value of the current index of the buffer array. To do this, the `PORTD` register is directly manipulated. By wiring the LSB of the DAC to pin 0, the second LSB to pin 1, and so on up to the MSB wired to pin 7, a WAV sample value can be directly put on PORTD, which automatically generates the correct analog voltage. When the index reaches the end of the array, it is reloaded. As a result, the buffer needs to be large enough that the MCU does not access the SD card too often to keep up, but also small enough that it can reload the entire buffer reliably between interrupts. Using this process, the original audio signal can be approximated (see Figure 1).

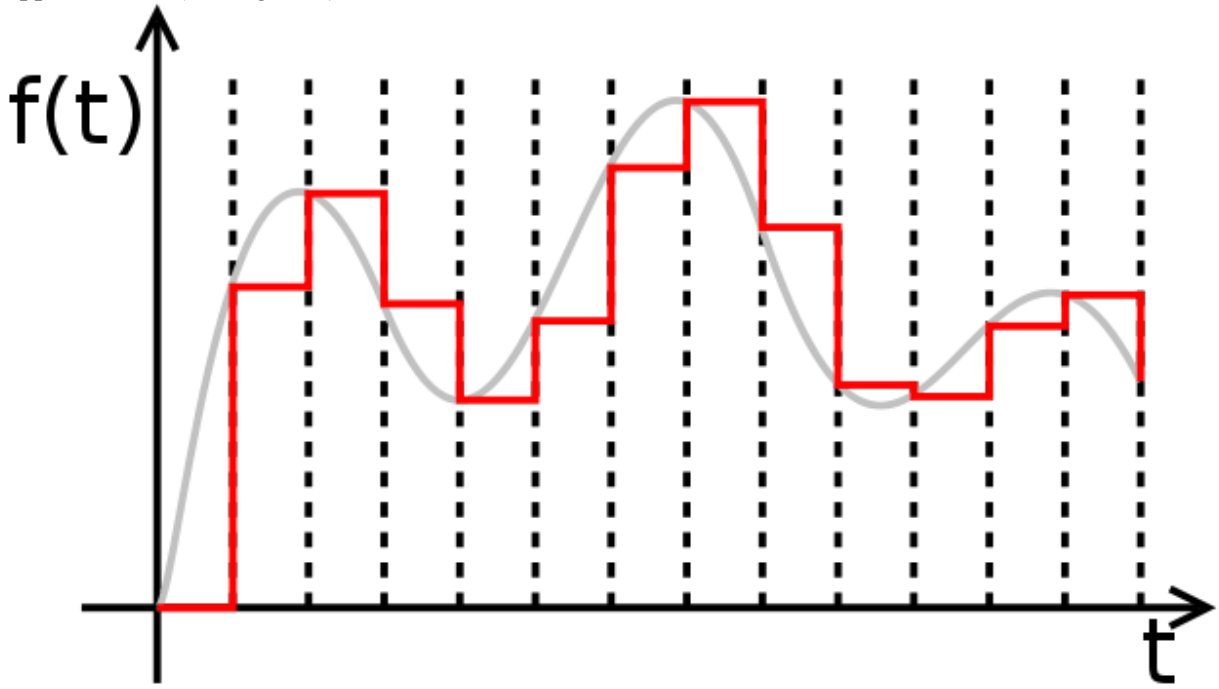



Figure 1. Audio Signal Approximation using Discrete Signaling

The 8-Bit Audio Player has 2 MCUs used: one "slave" to read from the SD card and update the DAC, and a master to coordinate which song to play, and by responding to user input through a rotary encoder and to control a TFT screen with the song progress and a list of songs. To communicate between MCUs, I2C is utilized. When the master wants to receive any information from the slave, it must send a request. Due to the way that I2C works (see Project 2.6.1), when a request is sent, the same *request routine* is run. A request routine is a function run when an I2C request is received. This means that the same data point is sent for every request, in a standard configuration. The request routine is simply a function. It is not, strictly speaking, an ISR. This means that many blocks of code can be efficiently run within the request routine, and blocking calls are considered acceptable.

To request multiple data points, a command code system is implemented. In this configuration, the master sends a code corresponding to a data point, which is placed into a buffer. Then, the master sends a request. Within the request routine, a subroutine is included for each point (see Figure 2).

**Figure 2. Code Data Correspondences**

| Code | Data Point |
|---|---|
| 0 | Song length |
| 1 | N/A |
| 2 | Song names |
| 3 | Number of songs |

The master MCU only has direct access to the display, the rotary encoder, and the I2C bus. Since it does not have access to the SD card, it cannot directly read the song names to put on the display. To read the song names, it requests the data from the player MCU. To do this, it first sends a value of 2, to place in the code buffer (see Figure 2). Then, it requests 32 bytes from the player, which reads the SD card, parses it into a character array buffer, and sends 1 character (byte) at a time to the master which reconstructs the data and parses it into a 2D array of characters. Since the master can only hold a finite number of song names in its 2 KB of memory, the player maintains a global index based on the last song sent. This means that on the next song name request, the next song is automatically sent.

In addition to a request routine, the player MCU has a *receive routine*. Unlike the request routine, the receive routine is an ISR, and it is called when data is written to the player. To aid the command code system, the first byte received is always the code. Some codes warrant a second byte (see Figure 3).

**Figure 3. Code Action Correspondences**

| Code | Action | Extra bytes |
|---|---|---|
| 0 | Changes song | 1 |
| 1 | Toggles Playback | 0 |
| 2 | N/A | 0 |
| 3 | N/A | 0 |

The code system is global, and not specific to the data direction. As a result, most codes are only valid in either Figure 2 (MISO), or Figure 3 (MOSI). The exception to this is the code 0, which, strictly speaking, corresponds to a song change. When a 0 is received, the `songChanged` flag is set, and a second byte is read, which is stored into the song number buffer. On the next iteration of the loop, the `songChanged` flag is picked up, and the song is set to the number in the song number buffer. When the song is changed, the master requires the length of the song each time, which is stored on the slave. As a result, the master sends a data request. Since the code 0 corresponds to a song change, an additional request code is not required; the data point requested immediately after a song change is always song length.

To select the desired song, a simple for loop is utilized. In the loop, an index is set such that the loop runs *song* times. In the loop are two important parts: a next file line, and a hidden file skipping line. In this configuration, since the loop is run a certain number of times, the next file is chosen the same number of times, skipping to the desired file. The index number is built in the same fashion which maps an index value to the filename, so a number can be easily linked with a filename from the display. The hidden file skipping line simply decrements the loop index if the file starts with "`._`".

When a song is selected, after the master updates the slave, it goes to the "now playing" screen (see Figure 4). This screen shows a music icon, which is stored as an array in program memory, to save RAM. It also shows the current song progress in seconds, the total song length, a progress bar, and the name of the track (see Figure 4).

Figure 4. Now Playing Screen

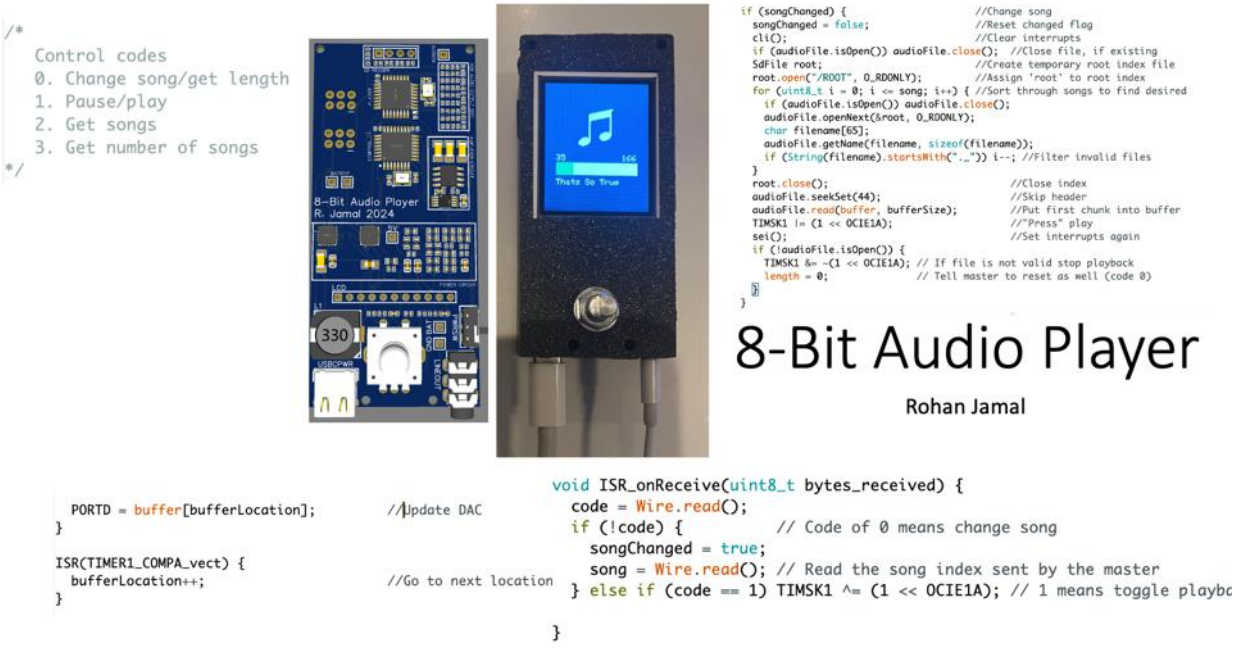
To display this screen, when the song is selected, the music icon is drawn from program memory (only once, which saves resources). The song track name is also drawn once. Then, a timer1-driven interrupt which triggers once per second is started, updating the song progress in real time. To display the progress bar, the map function is used. It simply maps the song progress from a value between 0 and `songLength`, to a value between 0 and 100. The progress bar is 100 pixels wide, so a green rectangle with the result of the map is drawn overtop of the white background bar.

On the now playing screen, when the rotary encoder is pressed, the master sends a code of 1 to the player, which toggles the OCIE1A bit of the TIMSK register. This bit enables or disables the execution of the ISR on the overflow of timer1. This effectively toggles playback. Since the MCU playing the music is independent from the one connected to the screen, the progress bar is updated independently as well; when playback is started, the master downloads the song length and starts a timer. It does not send a request with the song progress each second.

The PCB for the 8-Bit Audio Player includes a boost converter, as well as a battery charging IC. This allows the 5 V system to be used effectively with a 3.7 V lithium-ion battery, with charging over USB-C. The boost converter steps the voltage up from 3.7 V to 5 V. The battery charging IC regulates the 5 V from the USB-C connector to effectively charge the lithium battery.

## Media

Project video: https://youtu.be/TbR3IQ8DSpg



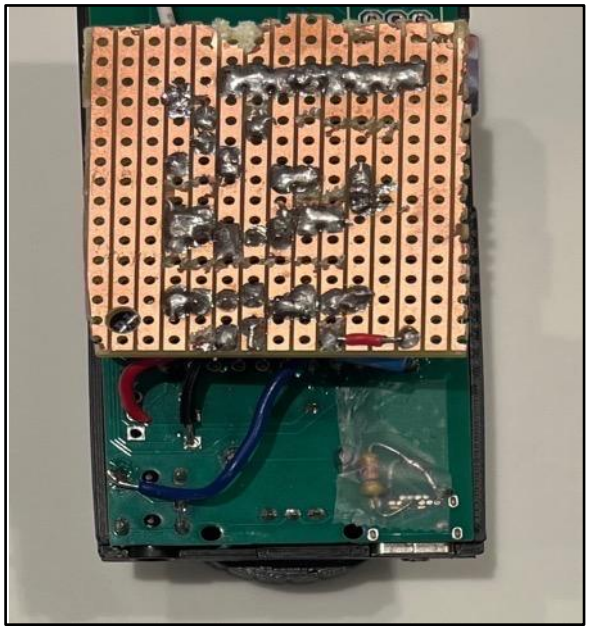

ISP Presentation Background


Left to Right: PCB, Assembled Device without Cover, Assembled PCB


New Year's Day Selected


New Years Day Playing (32/257 Seconds)

Device Side Profile (SD Slot)



PCB Rear (External Amplifier)

Rotary Encoder Handle

Device Ports (Bottom)

## Reflection

This has definitely been my favourite ISP thus far. Out of all my ISPs, I think this is my most functional one. My bike speedometer worked in the end, but it was never practical. This audio player is completely practical for sitting in one place whilst doing homework and listening to music. I learned a ton about how music is played in general, I learned a ton of strategies in code (it's just practice, I suppose), and I really feel that I pushed my PCB design during this project. Overall, I am super happy with the result as I feel that I accomplished just about everything that I said I would (more in some areas). The only "copout" that I ended up going with was 8 bits instead of 12, as all information I found pointed towards inaccuracies of the R/2R ladder beyond 8-10 bits. It also would not have made much sense from a port perspective.

I think that this is really not a testament to any skill but time management. This is something I normally struggle with, but I had my PCB done early enough that I was not scrambling at the last minute, and I had much time to debug and perfect my code and 3D design.

I know that I could have used an ESP32, or some other microcontroller to easily play the audio without an external DAC, but for me, the real satisfaction was building my own DAC from scratch, and working with the limited resources of the 328P

That being said, there are some things that I hope to fix: I think that if I get a chance, I will reorder the PCB, without the power circuit and use larger resistors and capacitors so I can surface mount them myself in order to address a few issues (such as lack of volume control and the absence of a functional onboard amplifier). Overall, this was a great project, and while thinking about my next ISP, I definitely have audio in mind. (My code and EasyEDA files are included in the GitHub and Oshwlab links in Reference).

# Project 3.4 AVR Assembly: Traffic Light

## Purpose

The purpose of the traffic light project is to teach a foundation for AVR assembly language. By coding an assembly program to control the sequence of a traffic light, basic assembly language techniques are introduced.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/Tasks.html#TrafficLight

Project GitHub: https://github.com/rohan-development/TrafficLightAssemblyIntroduction

## Theory

In Project 3.2 CHUMP, the concept of assembly language programming was introduced. CHUMP was a 4-bit processor, which restricted it to a maximum of 16 instructions. In assembly language, rather than use easy-to-understand syntax, such as `if()` statements and `for()` loops, the user is able to interact directly with the hardware, specifying a CPU instruction. Where 1 high-level instruction can translate to several [hundred] clock cycles, a line of assembly, or instruction, generally translates to 1 executed clock cycle. There are exceptions to this; some complex instructions can use up to 5 clock cycles. When working with a limited-processing-power microcontroller, such as the ATmega328P, this allows for extreme optimization of the hardware. This means that the same microcontroller can be used for more complex tasks.

It is important to note the internal memory structure of the ATmega328P before gaining an understanding of AVR assembly language. Figure 1 depicts the internal RAM structure of the ATmega328P. There are a total of 2048 8-bit registers (bytes) of SRAM, and a total of 256 other registers.

**Figure 1. ATmega328P RAM Structure**

| Reg # | Description | Address Range |
|---|---|---|
| 32 | GP Registers | 0x0000-0x001F |
| 64 | I/O Registers | 0x0020-0x005F |
| 160 | Extended Registers | 0x0060-0x0FF |
| 2048 | General SRAM | 0x0100-0x08FF |

The first 32 registers are the general-purpose registers. These are known as *working registers*. The working registers are used to hold temporary values, and to compute them, before they are either stored to memory using the `sts` instruction, or written to an I/O or extended I/O register. For example, there is no way to load an immediate (constant) directly into an I/O register, so the constant must first be put into a working register, and then the working register can be stored to the I/O register.

The next 64 registers are the I/O registers. The main difference between the standard and extended I/O registers are the way they are addressed. The standard registers occupy a lower set of addresses, which means that they can be accessed in a single clock cycle. The extended I/O registers take an extra byte to address, which means that they take a different set of instructions to access. The more commonly used registers, such as the PORT, PIN, DDR, and interrupt registers are in the standard register space, whereas configuration registers, such as the SPDR, which controls the SPI bus, is in the extended I/O space.

In AVR assembly, there are around 131 instructions. There are several instructions that perform similar functions (such as loading a register with a value). Each instruction has a specific type of accepted operand, which can be a certain type of register (see Figure 1), or an immediate (constant).

## Procedure

The ACES traffic light is a PCB with a green, a yellow, and a red LED onboard, arranged in a traffic light-like pattern. Each LED is connected to a pin, which allows each LED to be turned on or off.

| Parts Table | |
|---|---|
| **Quantity** | **Description** |
| 1 | Arduino Uno |
| 1 | RSGC ACES Traffic Light PCB |

In a high-level program, programming a simple sequence (such as green for 2 seconds, yellow for 1, and red for 2), would require each light to be explicitly set, as well as its own delay. In addition to being inefficient in terms of program space, it is inefficient in terms of resources used. A single `digitalWrite()` statement uses around 50 clock cycles, because it performs several redundant checks under the hood.

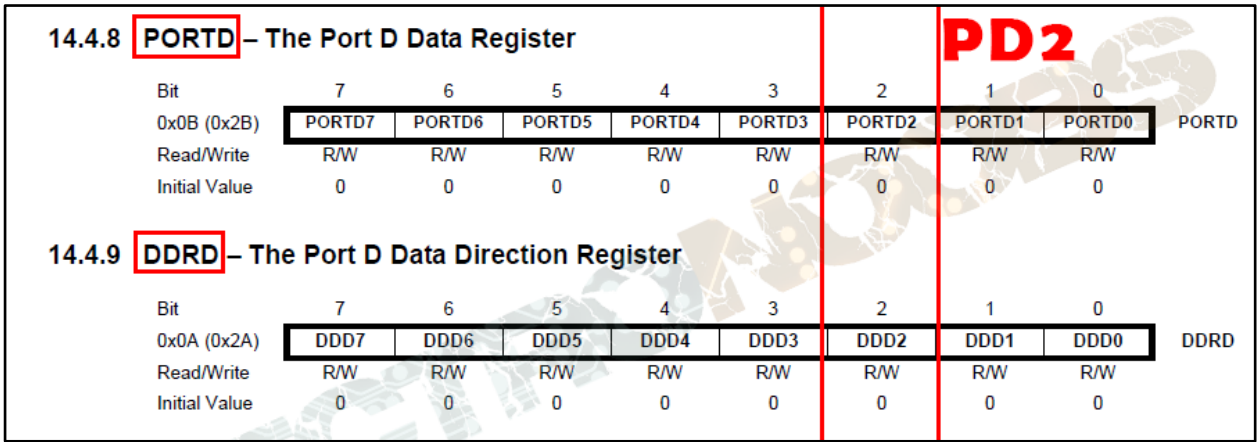



Figure 1. PORTD Pin Associated Registers

In assembly, `digitalWrite()` is replaced with 1-2 instructions, depending on the method used: in the first method, a bit mask corresponding to the bit location is placed into a working register, using the `ldi` instruction. Then, the working register is written to the desired port, using the `out` instruction. While this method uses 2 instructions, in can manipulate the entire port in 2 cycles. The second method is to set or clear an individual bit. To replace `digitalWrite(2, X)`, where *X* is either high or low, the statement `sbi PORTD,2` or `cbi PORTD,2` is used, respectively. This is because PD2 corresponds to digital pin 2 (see Figure 1).

The second high-level function that must be replaced in assembly for the traffic light project is the delay function. By definition, the delay function simply "wastes" clock cycles, so there is no real optimization to be had by performing this function in assembly. To delay for 1 second, 16 million clock cycles must go by, which means that 16 million instructions must be executed. For this, a register is simply decremented 16 million times in a loop. In an 8-bit CPU, since a loop can only have 256 iterations, multiple loops must run. For this, some computation must be run. There are 3 loops: one with 256 iterations, one with 43, and one with 82. Since the 256-iteration loop takes 2 cycles to execute, the total number of clock cycles is equal to: $256 \times 2 \times 43 \times 82 = 1805312$. At a clock speed of 16 MHz, this translates to approximately 1.13 seconds. The extra delay is due to the overhead of the loops.

The following is a breakdown of how the traffic light sequence is programmed in assembly, efficiently (see Reference for GitHub, or Code section): The first 6 lines are compiler directives that map each colour to a pin, and include a set of predefines that correspond specifically to the ATmega328P. The program starts on line 7, with an `ldi` instruction. This instruction is "load immediate," and it puts a constant into a working register (see Theory section). The stored constant is a bitmask that sets the pins attached to the red, yellow, and green led high. This bitmask is then stored to DDRB on the next line using the `out` instruction. Line 9 is simply a label for line 10, as it is a "branching point" which means that the program counter can be set to this point later on using a relative jump (or a regular jump). Line 10 puts the bitmask corresponding to the green LED into working register 16. It is the reset point of the LED sequence.

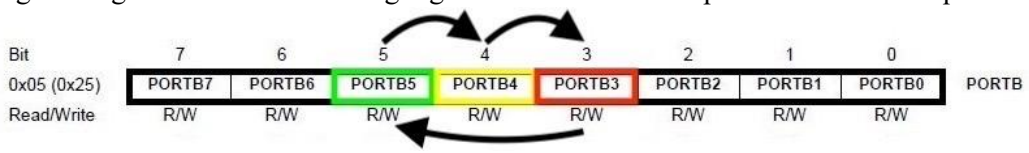



Figure 2. Green, Yellow, Red Pin Map and Bit Sequence

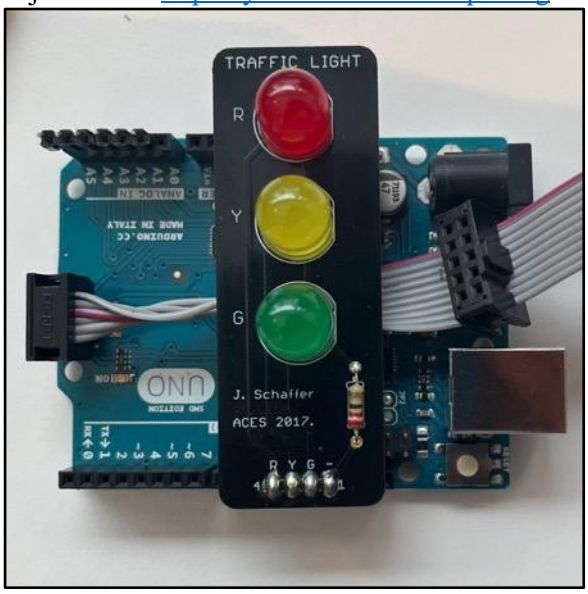
Since the yellow and red LEDs are simply the green LED shifted right once and twice respectively (see Figure 2), the program simply shifts working register 16 right, delays 1 second twice, by using the `rcall` instruction in conjunction with the delay1s function. Then, on each iteration, it uses the `cpi`, or compare instruction to check if it has been shifted beyond the red LED. If it has, the zero flag of the status register is set. This means that when a `breq` instruction is executed, it will branch. This branches to the reset point, and starts the sequence again. With the same logic when yellow, it only delays once.

## Code

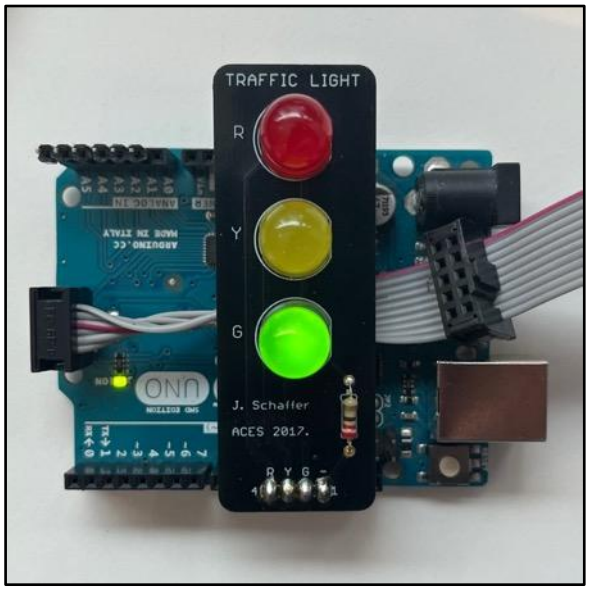
```
.include "m328pdef.inc"    ;Include 328P predefines
.equ GREEN = (1<<PB5)      ;Define green LED pin
.equ YELL = (1<<PB4)       ;Define yellow LED pin
.equ RED = (1<<PB3)        ;Define red LED pin
.equ PORT = PORTB          ;Define used port (DDR)
.equ DDR = DDRB            ;Define used port
ldi r16,GREEN|YELL|RED     ;Bitmask of 3 pins in r16
out DDR, r16               ;Put bitmask into DDR
reset:                     ;Label for reset point
ldi r16,GREEN              ;Put green into the bitmask
run:                       ;"Loop" label
out PORT,r16               ;Write out current colour
rcall delay1s              ;Wait 1 second
cpi r16,YELL               ;Check if YELL is in r16
breq yellow                ;Branch if YELL was in r16
rcall delay1s              ;Wait another second if not yellow
yellow:                    ;Label for when colour=yellow
lsr r16                    ;Shift colour down 1 bit
cpi r16,(1<<PB2)           ;Check if colours overflowed
breq reset                 ;If overflowed, reset colour
rjmp run                   ;If no overflow, loop again
delay1s:                   ;Delay 1 second function
  ldi  r18, 82             ;Load 82 into r18
  ldi  r19, 43             ;Load 43 into r19
  ldi  r20, 0              ;Load 0 into r20
L1:                        ;Loop 1 label
  dec  r20                 ;Subtract 1 from r20
  brne L1                  ;If r20 has a 0, go to L1
  dec  r19                 ;If not, decrement r19
  brne L1                  ;If r19 has a 0, go to L1
  dec  r18                 ;If not, decrement r18
  brne L1                  ;If r18 has a 0, go to L1
  ret                      ;Return (exit function)
```

## Media

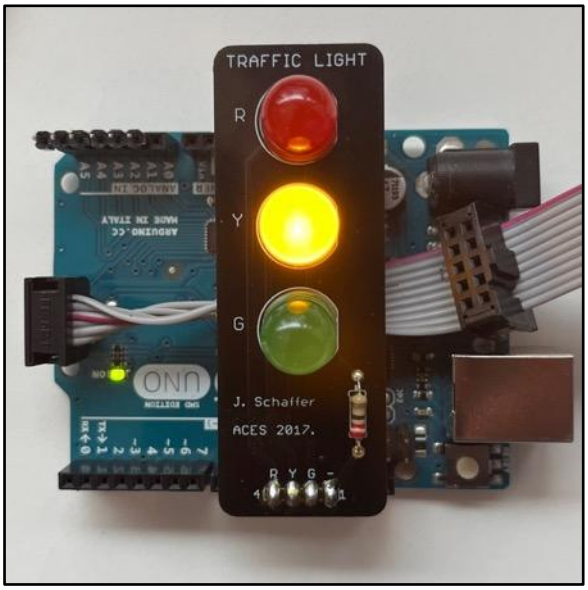
Project video: https://youtu.be/VmWGHp7-iGg



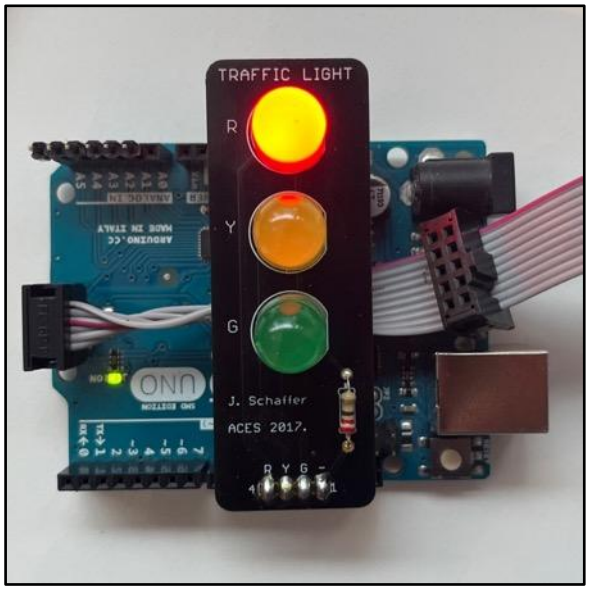

Traffic Light Build



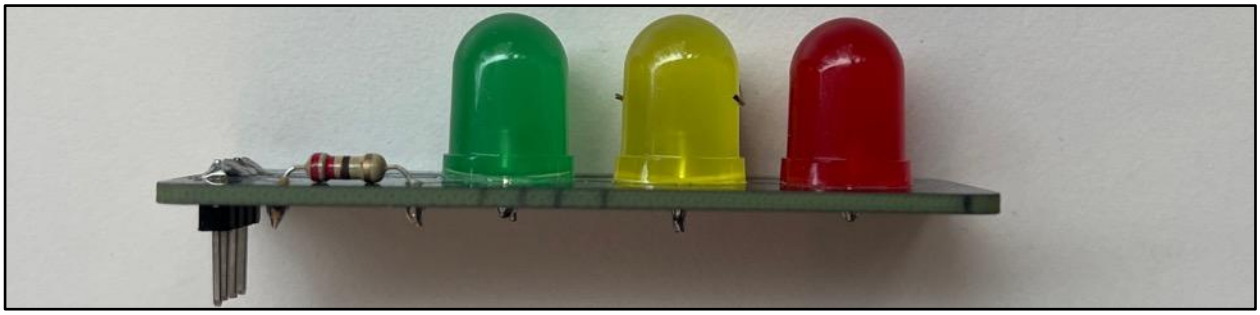

Traffic Light Green (2 Seconds)




Traffic Light Yellow (1 Second)




Traffic Light Red (2 Seconds)



ACES Traffic Light PCB Side View

## Reflection

I think that this was a great introduction to assembly coding. I learned some new techniques (like using a working register and shifting through it to affect certain pins) that I have no doubt will be helpful in the future. As I think of new ISPs, I am hoping that I do something involving assembly, although I'm not sure what that looks like yet, and I still have to experiment with it to determine my own skill level with assembly.

I will say that most of my time on this project was not spent building the circuit, writing the code, or even writing my DER and making the video. It was without a doubt, getting Microchip Studio to cooperate. And that is certainly saying something, because I had to record my video several times. The first time, it didn't record sound. The second time, it only recorded the back window, not the processor status and I/O status windows.

I spent a very long time trying to figure out why AVRDUDE wouldn't launch properly (eventually I figured out that it was because I was missing the AVRDUDE configuration file). I also spent a fairly long time trying to import my Microchip Studio code into word with the formatting. I never actually figured it out, so I just manually changed all the colours instead.

Overall, I learned a fair amount with this project, though I do still have much learning to do in terms of assembly, windows screen recording, and Microchip Studio. I look forward to the next assembly language project.

# Project 3.5 SAR ADC (Successive Approximation Register Analog to Digital Converter)

## Purpose

The purpose of the *SAR ADC* (successive approximation register analog to digital converter) is both to demonstrate how voltages are read by computers, and to map an unknown analog voltage to a 7-bit digital number using the successive approximation algorithm.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/SARADC/index.html

## Project 3.5.1 Overview and Clock

### Purpose

The purpose of the clock is to synchronize the operations of the SAR ADC. Since it uses successive approximation, multiple actions must take place, which warrants the use of a clock signal (for sequential logic).

### Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/SARADC/index.html#task

Project schematic: https://crcit.net/c/f705567608d5456ba69e7a610a942e96

https://dewesoft.com/blog/types-of-adc-converters

### Theory

When a computer or digital circuit needs to interact with "real world" signals or quantities (which are generally analog), a digital approximation of that signal must be created. To do this, an ADC is utilized. There are two main attributes of an ADC: resolution, and sample rate. Resolution refers to the precision of the approximation. A more precise (higher resolution) ADC generates an approximation closer to the original analog signal (see Figure 1). A greater sample rate correlates with fewer changes between samples, or a smoother curve, as exhibited in Figure 1.

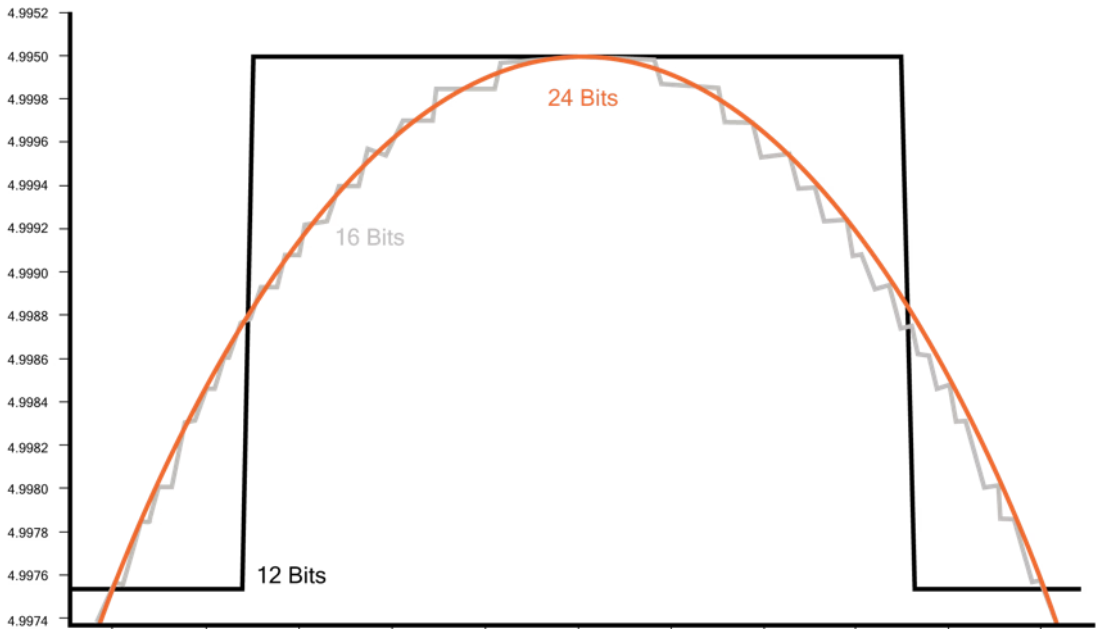



Figure 1. ADC Precision Comparison

## Procedure

There are several types of analog to digital converters. One of the most commonly used, however, is the successive approximation register (SAR) analog to digital converter (ADC). This type of ADC approximates an analog signal in several steps, starting with the MSB and working down the LSB.

To generate the approximation, an op amp is utilized as a comparator (see Figure 1). In this configuration, when the voltage presented on the non-inverting input (+ input) is greater than the voltage presented on the inverting input (- input), the output pin is high. When the voltage on the non-inverting input is less than the voltage on the inverting input, the output is low.

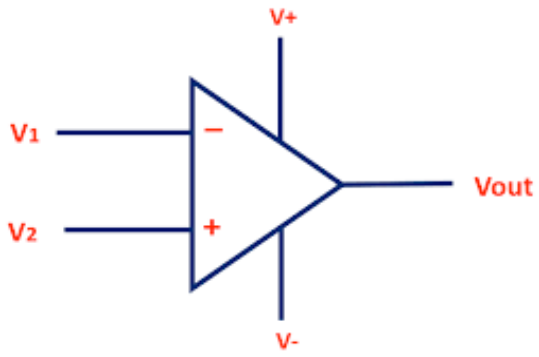



Figure 1. Op Amp in Comparator Configuration

In this configuration, a comparator can be used to determine whether or not an unknown voltage on the non-inverting input is above a threshold, which is set by the reference voltage that is present on the inverting pin (see Figure 1).

Since the sampling process does not happen instantaneously, and the voltage can fluctuate while the sample is being calculated, a *sample and hold* circuit is utilized (see Figure 2). A sample and hold circuit essentially stores the presented voltage for a set amount of time, in the case of the SAR ADC, the sampling duration.

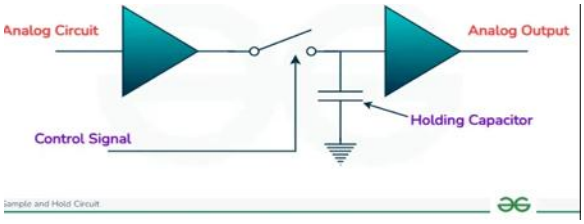



Figure 2. Sample and Hold Circuit

The SAR ADC utilizes a single comparator, with its reference voltage controlled by an R/2R DAC (see Project 2.1 and Project 3.3). On the first iteration, once the voltage is stored in the sample and hold circuit, the DAC has its MSB set and its other bits cleared. In a 5 V system, this sets the reference voltage to 2.5 V. The output of the comparator is fed into the output register, so if the voltage being sampled is greater than 2.5 V, a 1 is placed into the MSB of the output register. If the voltage is less than 2.5 V, a 0 is placed into the MSB of the output register. Additionally, the output register is tied back to the DAC through control logic. In this configuration, the DAC receives the already-calculated bits.

After the MSB has been calculated, the second MSB is calculated by setting the second MSB. This solves the second MSB, in a similar fashion to the MSB. This process is successively repeated, until each bit has been solved for.

Figure 3 depicts the successive approximation algorithm using a binary tree. In Figure 3, the sampled voltage is 5 V in a 9 V system.

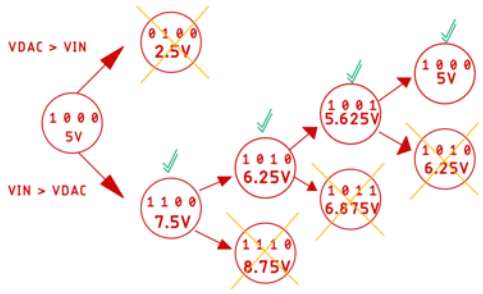



Figure 3. Successive Approximation Binary Tree

The binary tree (see Figure 4) is closely related to the successive approximation algorithm, as the SAR algorithm uses a *binary search algorithm*. A binary search algorithm involves starting at the MSB, and having a node for each bit, which allows for two decisions.

By utilizing a binary tree structure, the SAR ADC ensures that the sampling process executes in a predictable number of clock cycles. If the ADC had to cycle through each possible binary value, the total time to sample would vary greatly depending on the sampled voltage. Additionally, it allows the SAR ADC to operate relatively quickly. Some SAR ADCs sample at rates in excess of 10 MHz.

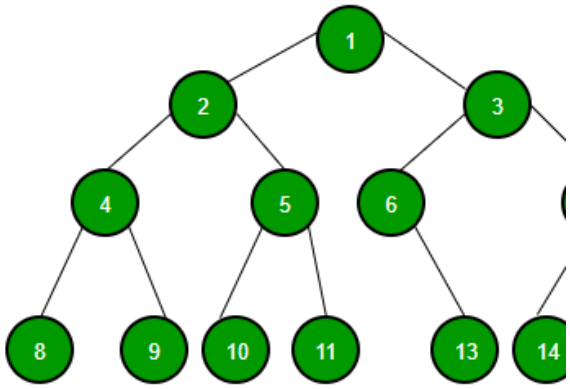



Figure 4. Generic Binary Tree

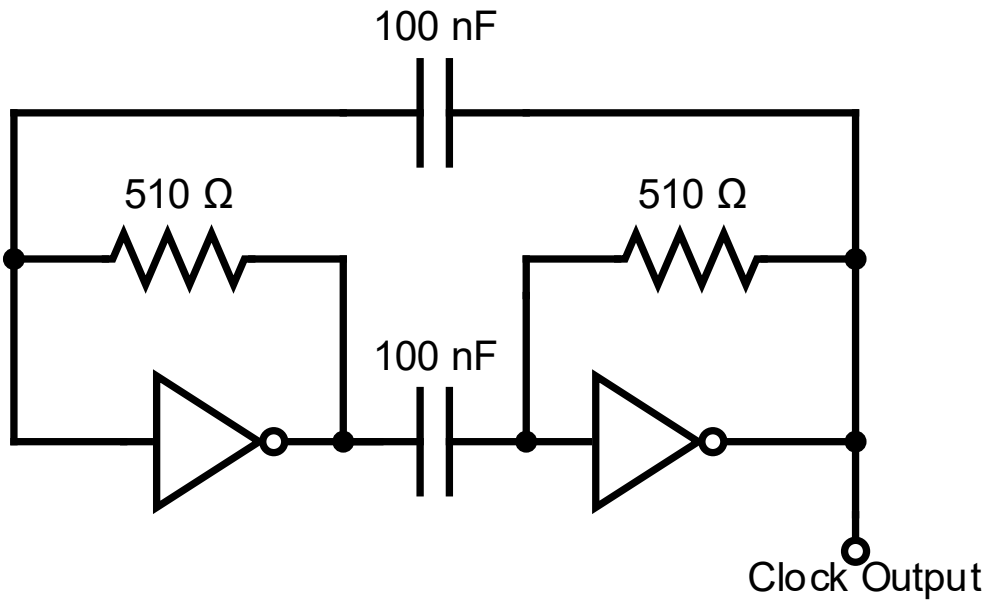
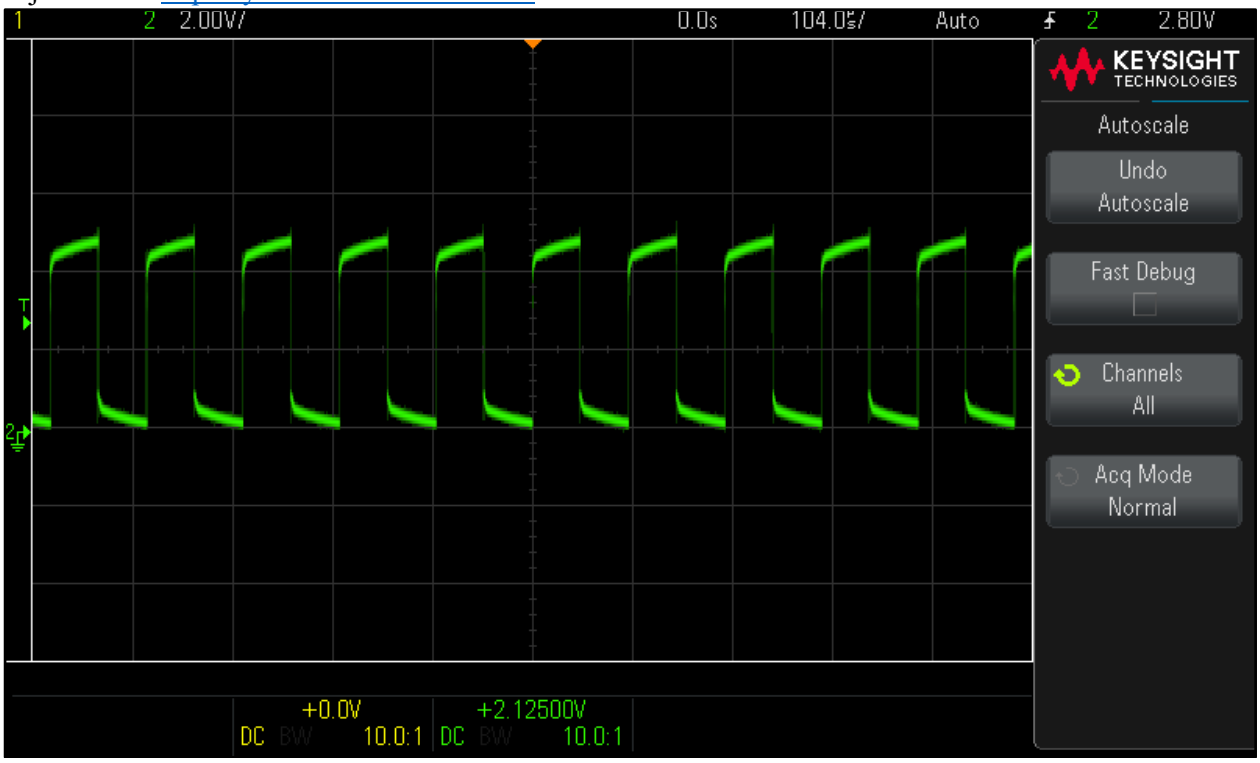



Figure 5. SAR ADC Clock Generator

The clock signal of the SAR ADC relies on an astable multivibrator built from NOT gates (see Figure 5). In this configuration, the resistors create a feedback loop between each NOT gate. As the capacitors constantly charge and discharge, a square wave is created. The clock speed of the SAR ADC is approximately 10 KHz, with a period of approximately 103 µs (see Media section).

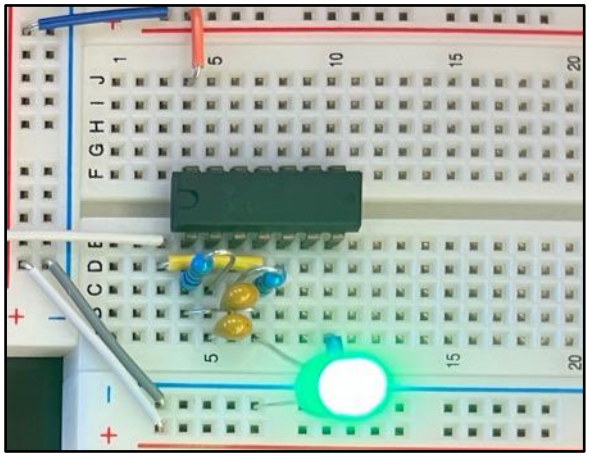
## Media

Project video: https://youtu.be/dIuDrUOZVYI




Oscilliscope Reading of Clock Circuit (Period of ~103 µs)


DMM Confirmation of ~10 KHz



Clock Oscillator Circuit from 74HC04

## Reflection

This is a very satisfying introduction to the SAR ADC. Conceptually, while I had a general idea of how successive approximation worked, I now have a solidified understanding. I think that it is not a difficult concept to grasp, in fact, I would argue that this is the simplest project we have had (conceptually) since grade 10. I will admit that building the circuit will likely prove to be slightly cumbersome and require much troubleshooting (good thing:) that I am looking forward to!

## Project 3.5.2 R/2R Ladder DAC

### Purpose

The purpose of the utilisation of an R/2R ladder DAC in the SAR ADC is to provide a variable reference voltage to compare the sample voltage to on each iteration of the successive approximation.

### Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/SARADC/index.html#DAC

### Theory

An operational amplifier, or op amp, is an extremely versatile electronic component that is made from resistors, capacitors, and transistors (see Figure 1). On use of an op amp is a comparator, which is a device that is used to compare to voltage levels. In this configuration, when the voltage presented on the non-inverting input exceeds that presented on the inverting input, the output of the op amp is high. Therefore, when a known threshold is presented on the inverting input, the level of an unknown voltage relative to the threshold can be tested. This principle is extremely important for the operation of the SAR ADC. In the SAR ADC, an unknown voltage is successively approximated by changing the threshold present on the inverting input.

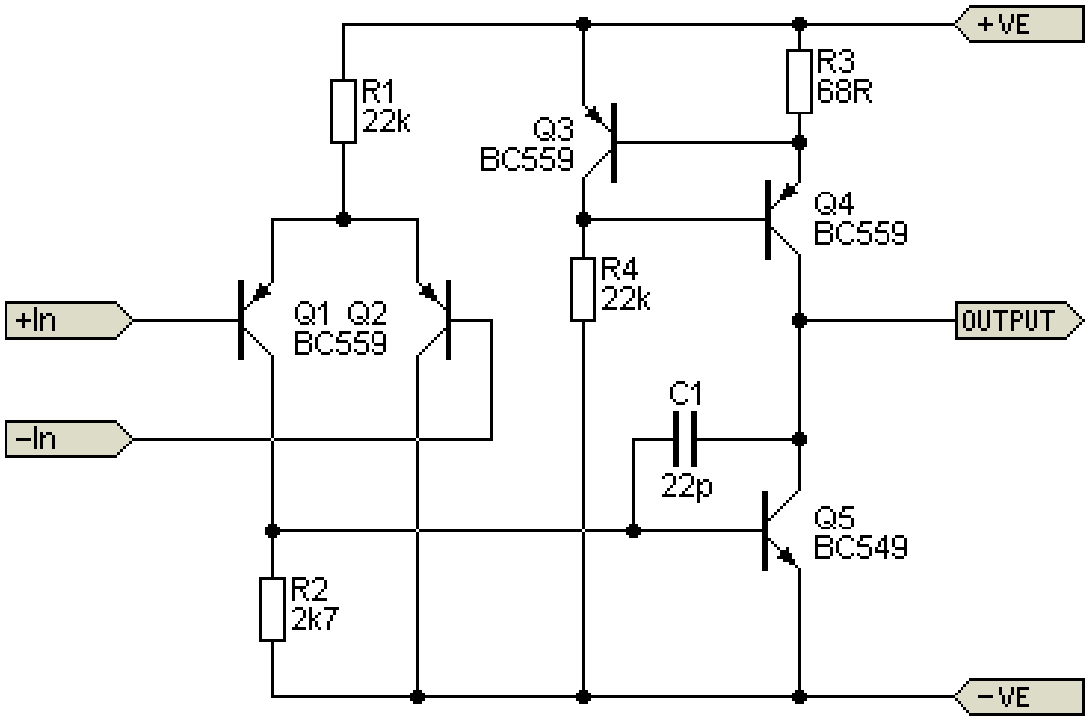



Figure 1. Operational Amplifier Internals

## Procedure

The LF398 sample and hold IC utilized in the SAR ADC (see Project 3.5.1:Procedure) retains inputs that fluctuate between 0 and 5 V. To do this, it requires a negative and positive power supply of 9 V. To obtain this voltage range, two standard 9 V power supplies are utilized. In this configuration, the two power supplies are wired in series, and a virtual ground is established where the positive and negative terminals meet (see Figure 1). The positive terminal of the first power supply is 9 V higher that the virtual ground, so it is positive 9 V. The negative terminal of the second power supply is 9 V lower than the virtual ground, so it is negative 9 V.

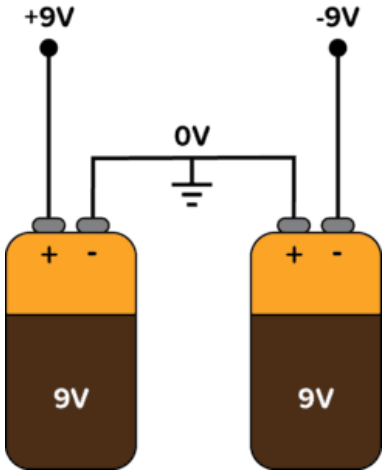



Figure 1. ± 9 V Power Supply

A supply range greater than 5 V is required because a buffer above and below the operating range is needed. This is because the LF398 is not a *rail-to-rail* component. A rail-to-rail component is a component in which the output can swing all the way to either the positive or negative (or ground) supply rail. The LF398 utilizes bipolar transistors, which consume some voltage. This means that without a dual supply of ±9 V, the held voltage would differ from the sampled voltage.

The sample and hold IC requires a storage capacitor to store the voltage that has been sampled for the ~1 ms that it takes for the voltage sample to be converted to a digital value. In the final stage of the SAR ADC, a 1000 pF capacitor is utilized; in this stage of the SAR ADC, however, a 100 nF capacitor is present. This is because a 100 nF capacitor holds the sampled voltage for longer, allowing the user to view the internal workings of the SAR ADC in real time. In the final version however, a quick charge and discharge is preferred since the sampling process completes in under 1 ms.

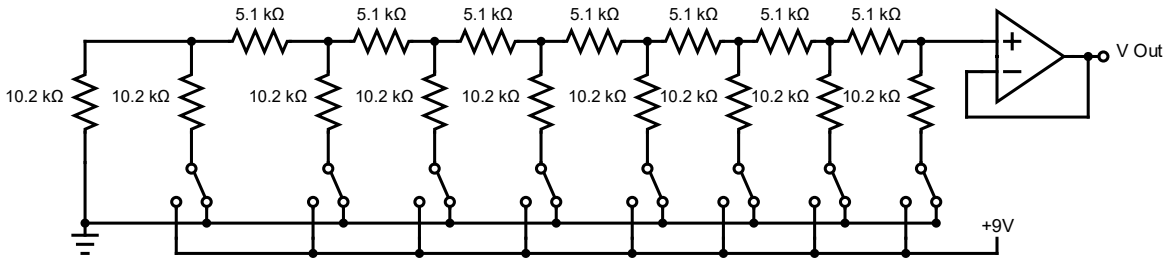



Figure 2. R/2R DAC with an Operational Amplifier-Driven Output

An R/2R DAC is a digital to analog converter that operates on the principle of voltage division (see Project 2.1 and Project 3.3) with binary-weighted inputs. Figure 2 depicts such a device, with. In this configuration, the MSB is represented by the switch on the right. This is because each bit to the left becomes progressively closer to the negative rail, bringing its overall contributed voltage closer to the negative rail. The resistor values are selected in a fashion that causes the inputs to be binary-weighted.

Figure 3 depicts the process that takes place in this configuration: the user dials the potentiometer to an unknown voltage, and stores it into the sample and hold IC by pressing the PBNO. Then, the user turns on the MSB. If the LED remains on, the user moves to the next switch, and repeats the process. If not, the user resets the MSB before continuing. This models the process that the eventual conversion process that will take place.

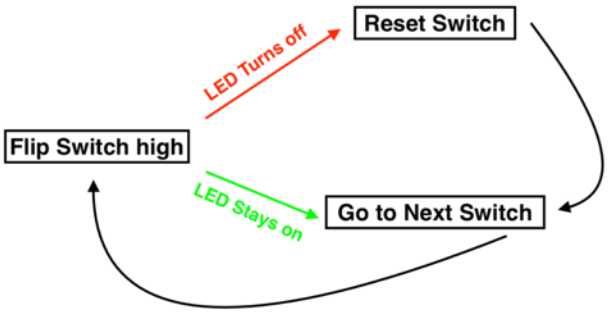



Figure 3. Manual SAR Process

Since the entire SAR ADC operates at 5 V (with the exception of the sample and hold IC and the comparator), a total of 3 voltage levels are required: 5 V, 9 V, and -9 V. Each 9 V rail is supplied by a discrete switching power supply.

The 5V rail, however, is supplied by an LM7805 voltage regulator (see Figure 4). In this configuration an LM7805 converts power from the 9 V rail to 5V by turning 4 V into heat.

This (inefficiently obtained) 5 V is then distributed to the parts that require it. The 9 V power supply can still be used in parallel for the comparator and the sample and hold IC.

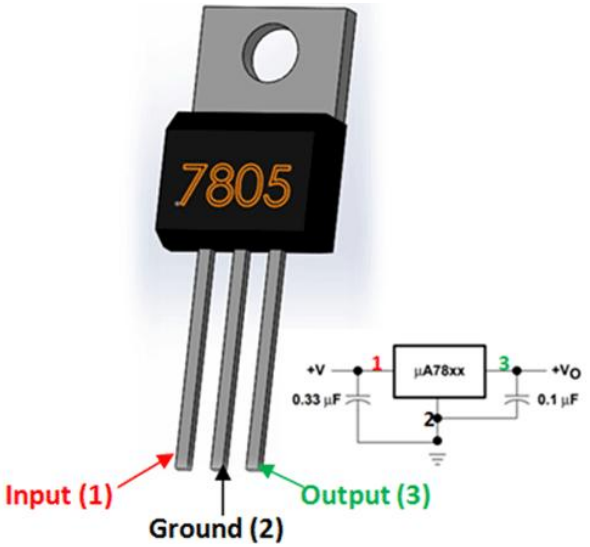



Figure 4. Manual SAR Process

This stage of the SAR ADC is asynchronous, meaning that it does not rely on a clock signal. The output of the R/2R DAC is fed into the sample and hold IC, who's output is used as the lower threshold of the comparator (see Theory section). This stage of the SAR ADC allows the user to manually sample and convert a voltage using a DIP switch bank, a potentiometer, a PBNO, and an LED.

The LM393 has an *open-collector* output. An open-collector means that the output can only sink current, and is unable to source current. To interface with digital logic circuits, the output is pulled high (see Figure 5).

There are several advantages of an open-collector system: firstly, it can interface with different voltage levels. For example, the comparator can operate at 5 V but pull up the output to easily operate at 3.3 V. Additionally, multiple open-collector outputs can easily be wired together to function with AND gate logic.

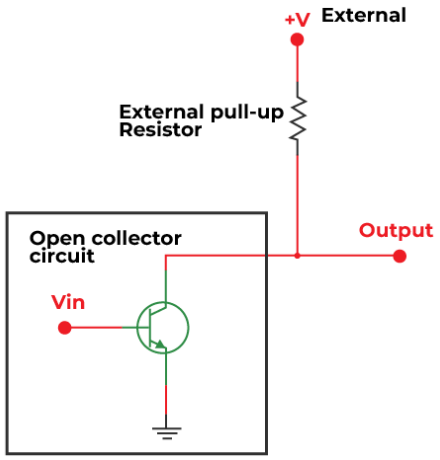
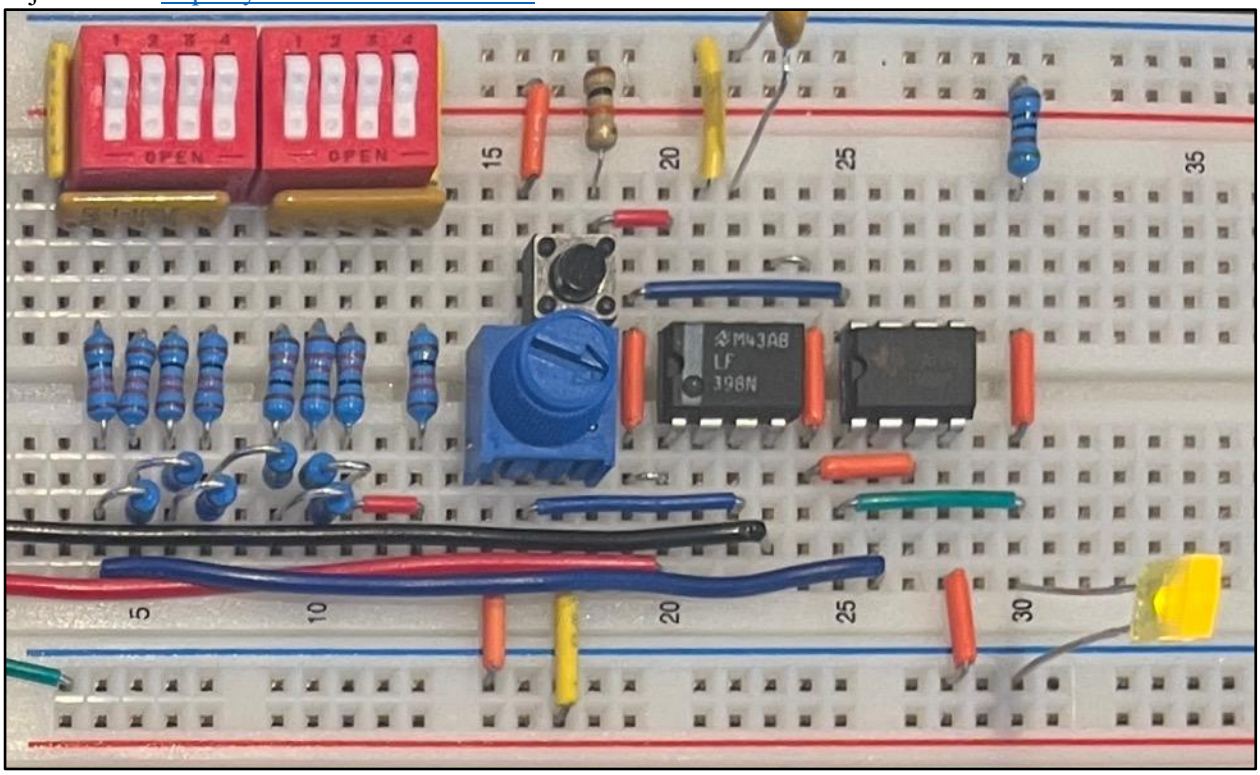


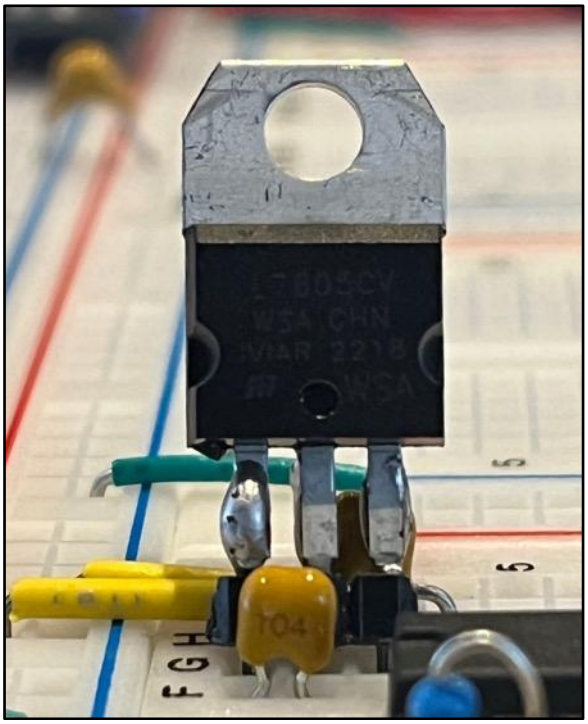

Figure 5. Open-Collector Output Configuration

## Media

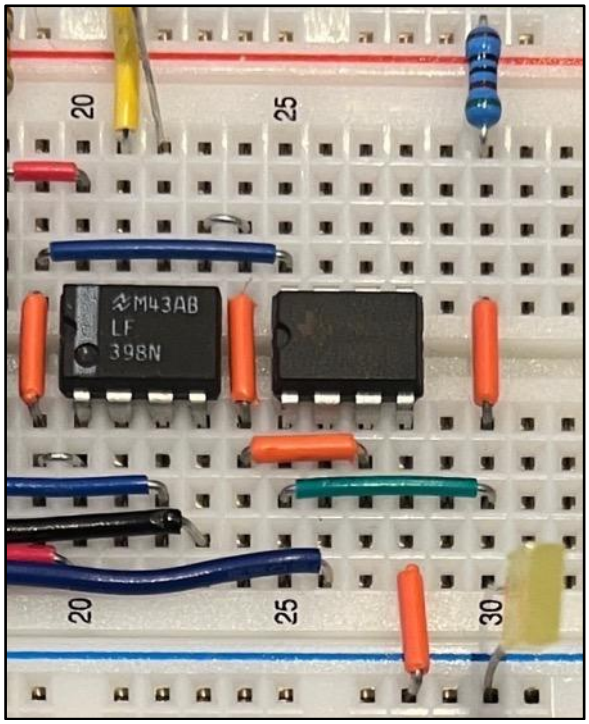
Project video: https://youtu.be/KrAzBvUE23o




SAR ADC R/2R DAC, Sample and Hold IC, and Comparator



LM7805 Voltage Regulator and Decouling Caps




Comparator and Sample and Hold IC Close-up

R/2R Ladder Closeup

## Reflection

This has been an okay stage of the project. I feel that this stage could have been combined with the overview and clock, as I wrote about much of the R/2R ladder and comparator stage within the overview. Other than that, I am pretty satisfied with the way that the analog input works. I am able to replace the SAR logic with my own brain by flipping the switches which is excellent for understanding how it works. I also was introduced to negative voltage in this project. Even though we were supposed to include negative voltage in the first R/2R project, I never did, but now I understand it.

I also am really appreciating the importance and versatility of the R/2R ladder circuit. In grade 11, I thought it was kind of cool, but not necessarily a practical or useful circuit. At this point, I have used it in two additional projects: my ISP (which relied on it very heavily) and the SAR ADC (which also relies on it quite heavily). Overall, my prototype seems to work quite well thus far, and I'm excited to finish the project!

## Project 3.5.3 SAR ADC Completed

### Purpose

The purpose of the completed SAR ADC is to convert analog voltages to a discrete digital value between 0 and 127. The current voltage being read can also be accessed over an I2C bus; this ADC is capable of acting as an I2C peripheral in both binary and voltage-reporting mode.

### Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/SARADC/index.html#Completed

Build credit: https://hackaday.io/project/181826-homemade-successive-approximation-register-adc

Original author: Mitsuru Yamada

### Theory

One of the main measures of an ADC's processing abilities is its sample rate. The sample rate is the maximum frequency at which the ADC can take samples of its input. The SAR ADC has a clock frequency of 10 KHz, and each sample takes 8 clock cycles, the sample rate is given by:

$$f = \frac{10\ \text{KHz}}{8\ \text{Samples}} = 1.25\ \text{kS/s}$$

This means that in 1 second, the SAR ADC takes approximately 1250 samples of the voltage presented on its analog input. This sample rate can easily be increased by increasing the clock frequency. While changing the bitness (see Project 3.2.5) of an ADC requires major changes in the wiring and components of the circuit, the clock frequency requires little change, as long as the components can handle a higher clock frequency.

Figure 1 depicts the difference between resultant wave functions with progressively higher sample rates. Increasing the sample rate even with a low bit-depth ADC can allow its output to much more closely resemble the original waveform (see Figure 1).

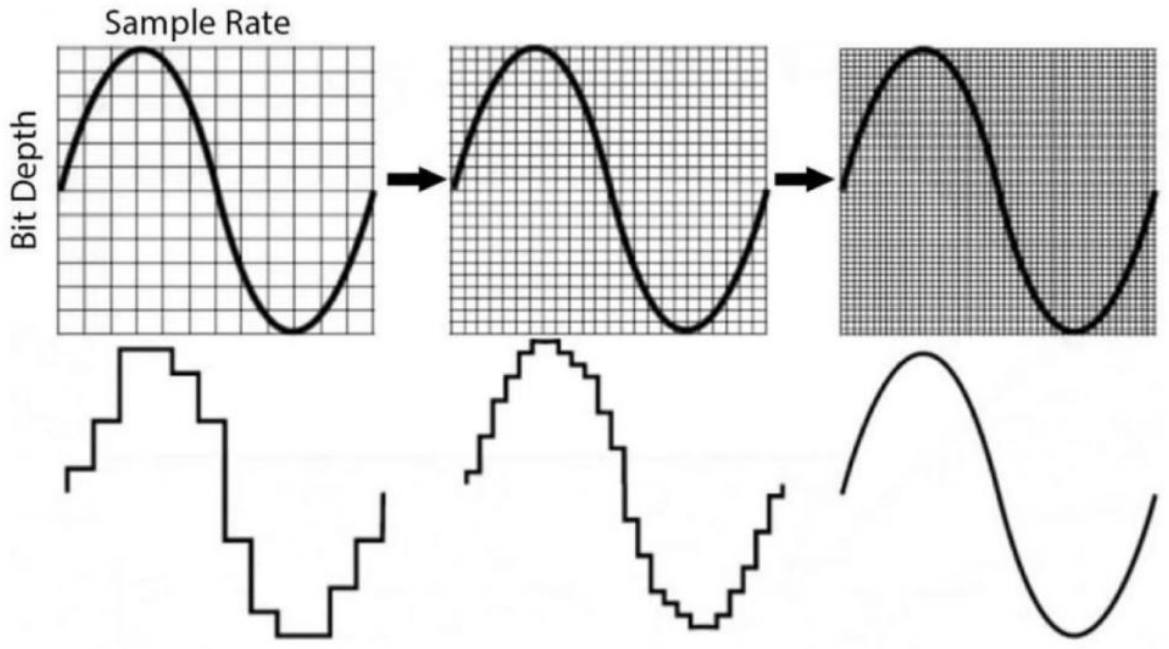



Figure 1. Sample Rate Increase vs Bitness Increase

## Procedure

The sample rate of 1.25 KHz is relatively fast, although it is not nearly fast enough for many audio-based applications which would require a minimum frequency of ~16 KHz. To calculate the maximum theoretical clock frequency that the SAR ADC could support while maintaining proper operation, the frequency-limiting IC must be found. The logic gates and flip-flops have propagation delays in the range of ~10ns, whereas the sample-and-hold IC has a propagation delay of ~8 µS with a 1 nF capacitor (see Figure 1). Assuming the SAR takes around 2 µS to compete the successive approximation, 1 sample would take a total of 10 µS. This translates to a theoretical sample rate of 1 MHz, or a clock frequency of 8 MHz.

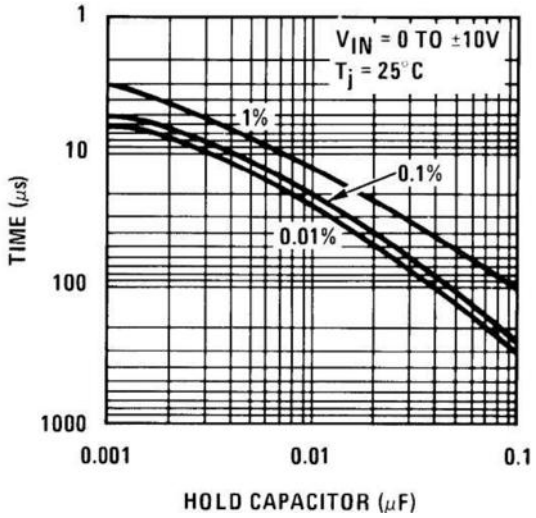



Figure 1. LF398 Acquisition Time

The SAR ADC supports two modes: manual, in which the output is latched and only updated when the sample button it pressed, and automatic mode, in which the output constantly updates to match the input voltage, at a sample rate of 1.25 KHz.

In this stage of the SAR ADC, the manual input to the R/2R Ladder is replaced with AND logic (see Figure 2). In this configuration, 1 input of the AND gate is connected to a D flip-flop, while the other is connected to a bank of cascaded D flip-flops. These flip-flops (not depicted in Figure 2) are cascaded from MSB to LSB, with the MSB connected directly to power. This means that on the first clock cycle, only the MSB has the chance for its output to be high. Then, on each successive clock cycle, the next bit's cascaded D flip-flop goes high, allowing it to update bit-by-bit.

The data input of the D flip-flops depicted in Figure 2 is connected to the comparator, and the clock is connected to the next bit's cascaded D flip-flops, which causes it to latch to the comparator's output. This is how the SAR ADC sets each bit to the correct value; when the bit's threshold is exceeded, the comparator is high, and latched to the appropriate flip-flop. This configuration automatically updates the R/2R ladder for the next iteration.

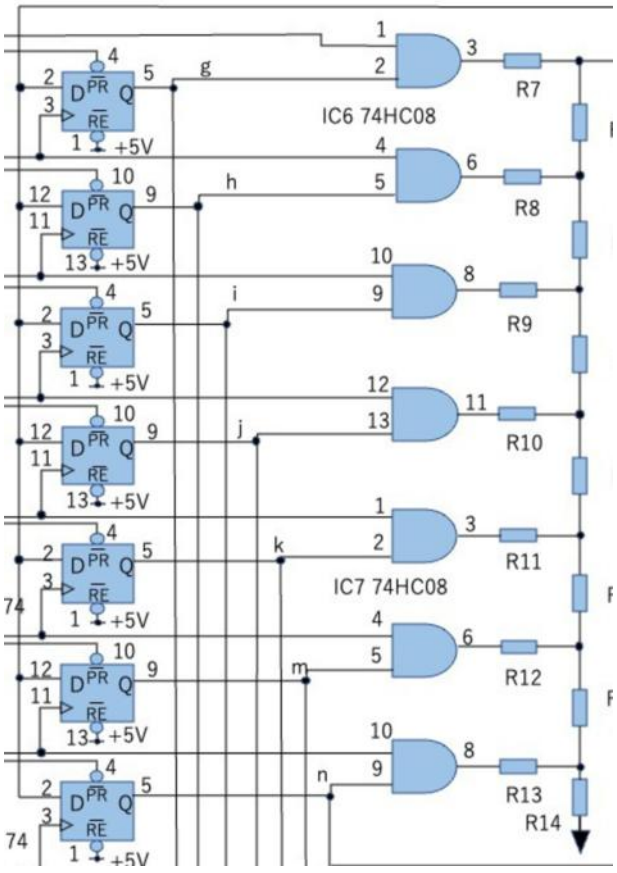



Figure 2. R/2R Ladder Inputs

This version of the SAR ADC is I2C compatible, meaning the sampled value can be accessed over an I2C bus. This is a direct result of the addition of an Arduino Nano in I2C slave mode, creating a smart I2C network (see Figure 3). In this configuration, the Nano has the outputs of the SAR ADC connected to digital pins 0-6, which is most of PORTD. Then, when the I2C master requests the data, it downloads the value in PIND, reading the entire SAR ADC output. The data can be requested in 3 different ways: 7-bit binary mode, 8-bit binary mode, or floating-point voltage mode.

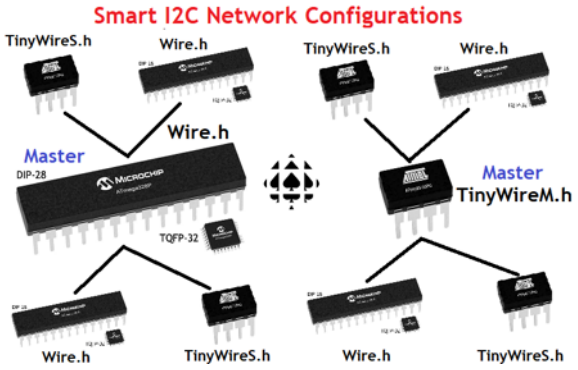



Figure 3. Smart I2C Network

7-bit binary mode sends the raw data, because the value read from PIND is an 8-bit value, but the MSB is not utilized, making it a 7-bit value. In 8-bit binary mode, the slave automatically bit shifts PIND left 1 place before sending it. This is because 8-bits is a standard number of bits, while 7 is not. By automatically converting the value before sending it, the master can use the raw received data, saving it clock cycles. When floating-point voltage mode is active, the slave automatically converts the binary value to a voltage value between 0 and 5 V with the following equation, where S is the 7-bit sample:

$$V = \frac{S}{127} \times 5\ \text{V}$$

I2C is an 8-bit protocol, while a float is 32-bit datatype with a distinct structure (see Figure 4). A float is interpreted in a similar fashion to scientific notation. The MSB is the sign, with a 0 corresponding to a positive value, and a 1 corresponding to a negative. The next 8 bits are the exponent, which is how many times the mantissa must be shifted left or right. Finally, the mantissa is the actual value, with 1 leading bit.

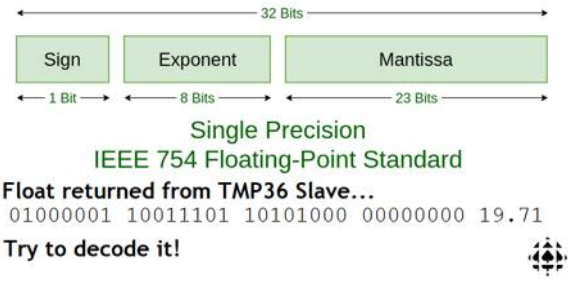



Figure 4. Single-Precision Float Structure

In order to send this value of I2C, a *union* is used. A union is a C programming structure in which 2 variables occupy the same address in memory. In this configuration, a 4-byte chunk of RAM can be written to as an array of 8-bit values, but then read from (and interpreted as) a floating-point decimal.

The slave utilizes a union to convert the float to a 4-location array of 8-bit values. It then sends these bytes over I2C, and the master has a union to reassemble this array into the floating-point number (see Figure 5). The master then prints the voltage over the serial monitor, allowing the value read to be graphed as a function of time.

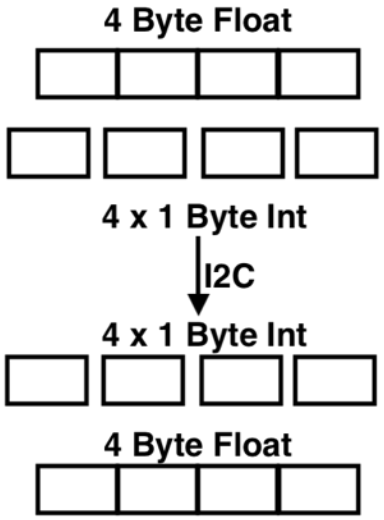



Figure 5. Float I2C Send Structure

To request multiple data points, a command code system is implemented. In this configuration, the master sends a code corresponding to a data point, which is placed into a buffer. Then, the master sends a request. Within the request routine, a subroutine is included for each point (see Figure 6).

| Figure 6. Code Data Correspondences | |
|---|---|
| **Code** | **Mode** |
| 0 | 7-Bit |
| 1 | 8-Bit |
| 2 | Floating-Point Voltage |

While accessing this SAR ADC over I2C does not have any major benefits over using the built in ADCs through `analogRead()`, it is simply a proof of concept. A higher frequency, higher bit-depth SAR ADC can be built in a similar fashion. In this configuration, the output could be 10 bits or more. Utilizing the main MCU to read from this SAR ADC is not practical; it would consume too many I/O pins. With an embedded I2C controller (an Arduino Nano, in this case), only two pins are consumed, and some data processing can be automatically done (see Figure 7).

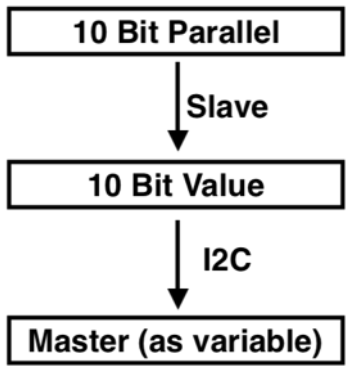



Figure 7. Float I2C Send Structure

It also allows for general expansion of the master. If extra EEPROM is needed, the master can offload the data to the slave's built in EEPROM. If extra (slower speed) RAM is needed, the master can store it in the slave's extra RAM.

Additionally, multiple ADC pins can be read by a singular I2C controller. In this configuration, the input of the SAR ADC is connected to an analog multiplexer, with several input pins, and a select line is controlled by the slave (see Figure 8).

The slave then expects a byte to be sent which corresponds to the ADC number (equivalent to A0-A5), and appropriately configures the multiplexer select line, reading from the appropriate input pin.

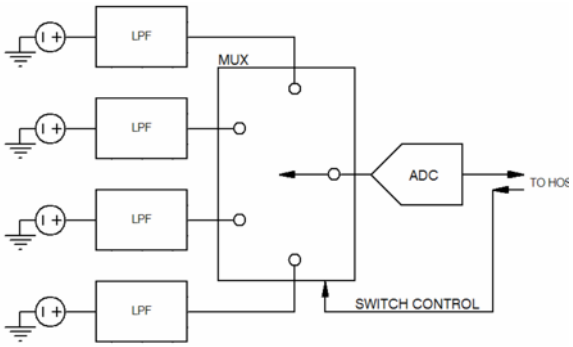



Figure 8. Multiplexed ADC

## Code

### *Master*

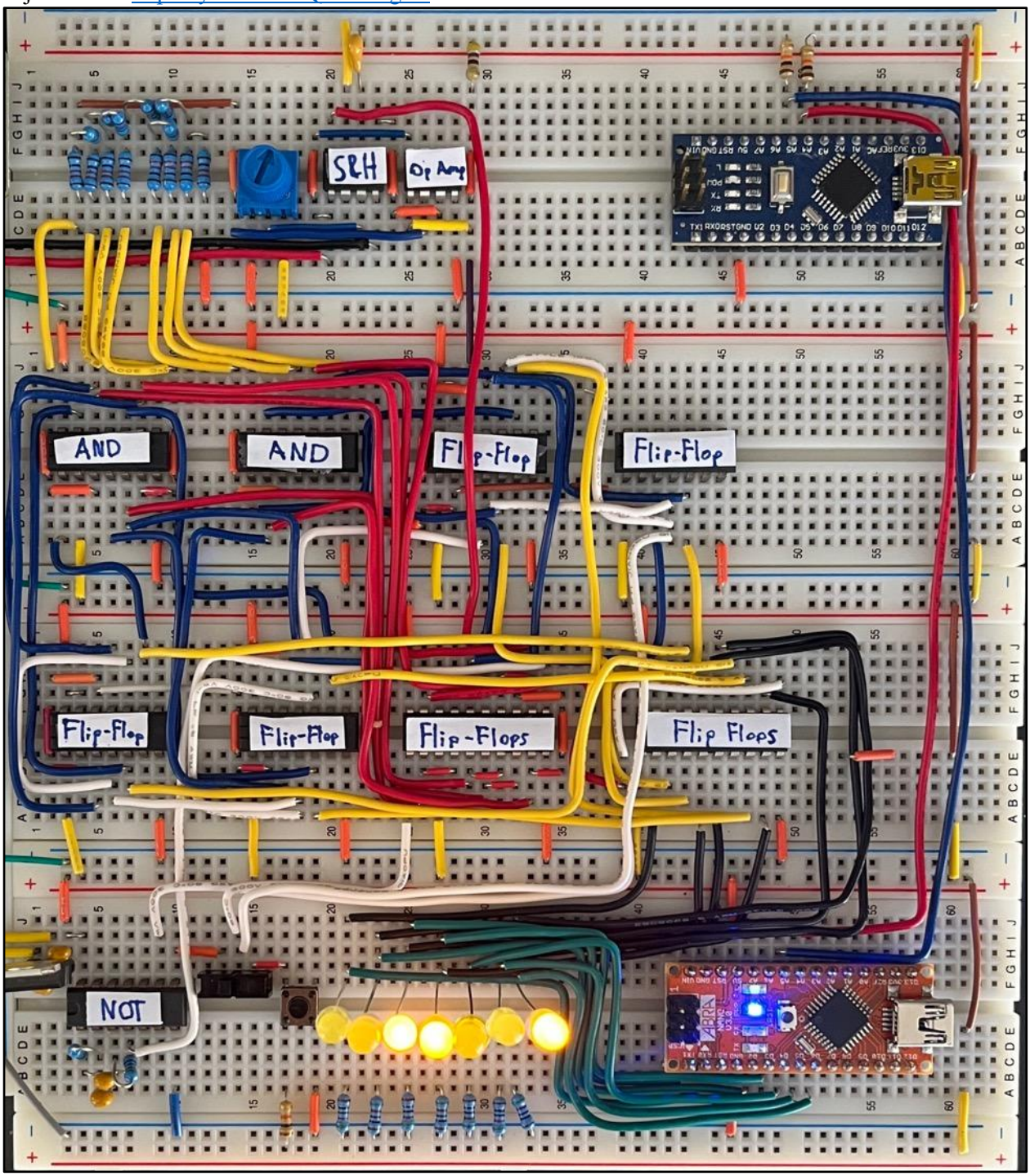
```
// PROJECT   : SAR Master
// AUTHOR    : R. Jamal
// PURPOSE   : To plot the I2C-compatible SAR ADC readings
#include <Wire.h>             //Include Wire for I2C
#define ADDR 0x20             //Slave I2C address
void setup() {
  Wire.begin();                 //Initialize I2C bus
  Serial.begin(9600);
}
void loop() {
  Wire.beginTransmission(ADDR); //Begin transmission to slave
  Wire.write(2);                //Request voltage on request
  Wire.endTransmission();       //End transmission
  Wire.requestFrom(ADDR, 4);    //Request 4 bytes
  while (!Wire.available());    //Wait for data to become available
  union b2f {                   //Union to receive float
    uint8_t b[4];               //8-bit wide array of 4
    float f;                    //Float componenent
  } data;                       //Name of union
  data.b[3] = Wire.read();      //Reads first byte
  data.b[2] = Wire.read();      //Reads second byte
  data.b[1] = Wire.read();      //Reads third byte
  data.b[0] = Wire.read();      //Reads fourth byte
  Serial.println(data.f);       //Print the received voltage
  delay(10);                    //Wait 10 ms per reading
}
```

### *Slave*

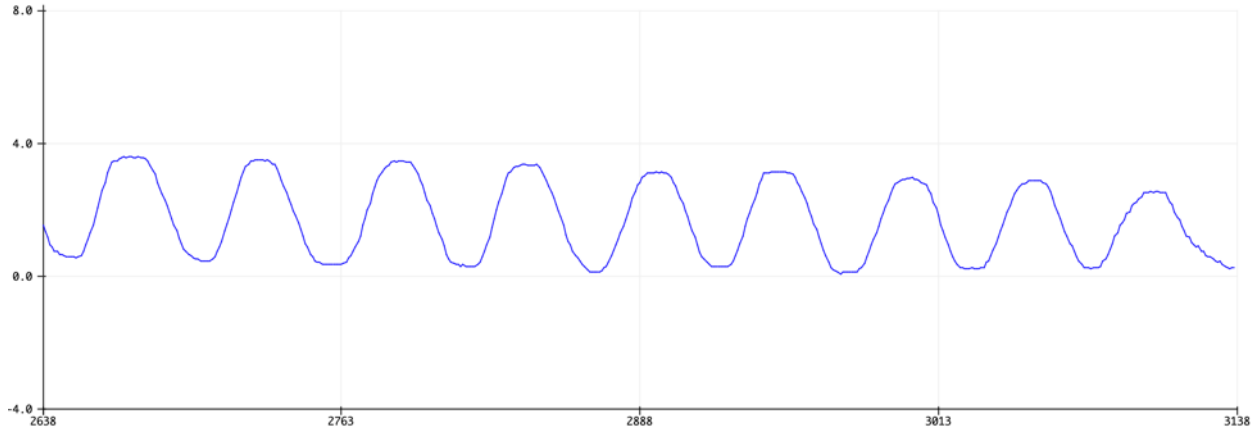
```
#include <Wire.h>                       //Include Wire for I2C
#define ADDR 0x20                       //Slave I2C address
uint8_t code;                           //Buffer for datatypes
/*     Codes:
   0 - Binary 8-bit
   1 - Binary 7-bit
   2 - Full voltage
*/
void setup() {
  Wire.begin(ADDR);                     //Initialize I2C
  Wire.onReceive(ISR_receive);          //Define receive handler
  Wire.onRequest(ISR_request);          //Define request handler
}
void loop() {}                          //Nothing to do
void ISR_receive(uint8_t bytes) {       //Receive handler
  code = Wire.read();                   //Receive datatype definition
}
void ISR_request() {                    //Request handler
  if (!code) Wire.write(PIND << 1);     //Report 8-bit value
  else if (code == 1) Wire.write(PIND); //Report 7-bit value
  else {
    union b2f {                         //Union to send float
      uint8_t b[4];                     //8-bit wide array of 4
      float f;                          //Float componenent
    } data;                             //Name of union
    data.f = (float(PIND) / 127) * 5;   //Calculate voltage
    Wire.write(data.b[3]);              //Sends first byte
    Wire.write(data.b[2]);              //Sends second byte
    Wire.write(data.b[1]);              //Sends third byte
    Wire.write(data.b[0]);              //Sends fourth byte
  }
}
```

## Media

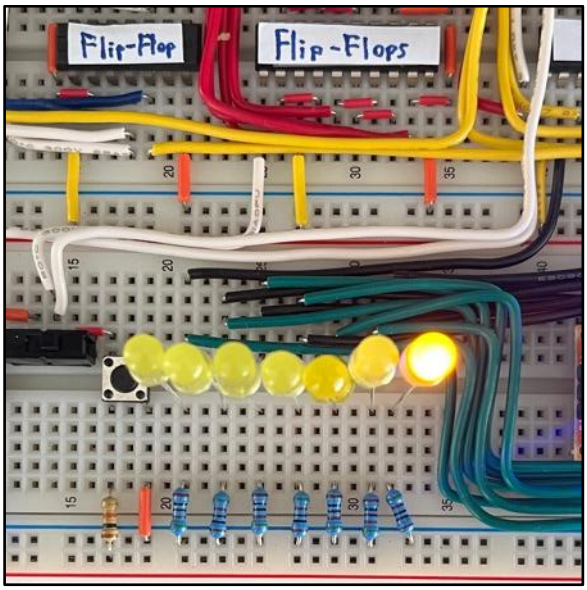
Project video: https://youtu.be/NQ83lcEegKc



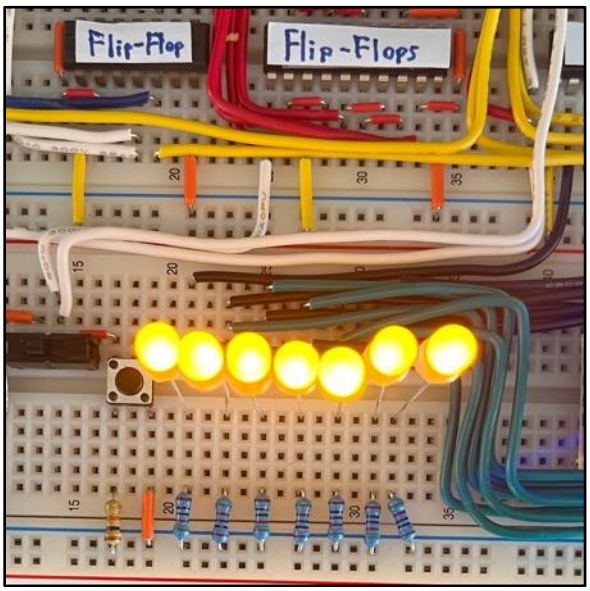

SAR ADC Entire Build




Serial Plotter when Potentiometer is Spun Back and Forth in Floating-Point Voltage Mode




SAR ADC Output at ~2.5 V




SAR ADC Output at ~5 V

## Reflection

This has actually been quite a good project. At the beginning, I must say that I was somewhat skeptical, and not sure whether or not it would be a project that I thoroughly enjoyed. Like CHUMP, it was my addition to the project that is what really sparked my interest. When I integrated the project with a microcontroller, I began to see how cool it really was. I was mostly just surprised to see how accurate it was. When I connected my DMM to the input, and put the slave in floating point voltage mode, they showed the exact same value which was pretty cool. I also was really surprised to have found a use for unions so quickly. When we learned about them in class, I did not think I would have the chance to use them so quickly, but my project certainly would not have been possible without their use.

# Project 3.6 Medium ISP: 32 by 32 RGB Matrix

## Purpose

The purpose of the 32 by 32 RGB matrix is to display the live image from a camera on a collection of 1024 standard (non-addressable) RGB LEDs using a constant current driver, advanced analog demultiplexing, dual RAM chips, and a microcontroller for video decoding.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI3M/2324/ISPs.html#logs

Project proposal: http://darcy.rsgc.on.ca/ACES/TEI4M/2425/images/RJRGBWall.png

## Theory

A modern display consists of millions of small squares, whose colour can be manipulated, called pixels. A pixel consists of a red, green, and blue (RGB) part. Each part can have its brightness precisely controlled. By mixing these three colours together, nearly any colour on the visible spectrum can be produced. Through precisely adjusting the colour of these pixels in a controlled manner, a complex image can be shown. In this configuration, brightnesses are represented as numbers, often between 0 and 255 (8-bits). This means that a typical pixel requires 24 bytes to be expressed.

Since even a relatively low number of pixels on the outside leads to a large number of pixels on the interior (for example a 50 by 50 display would yield a pixel count of 2500), each pixel cannot be controlled individually; an incredibly complex microcontroller would be required.

One of the ways that displays can control up to millions of pixels at a time is through *active matrix addressing* (see Figure 1). In this configuration, the rows and columns are arranged in a matrix. Each pixel has a thin-film-transistor (TFT) acting as a switch, a capacitor, and a voltage-controlling TFT. When a row is activated, the TFT turns on allowing charge to flow from the *data line* (column) to the pixel capacitor. The capacitor discharges into the voltage-controlling TFT, converting the voltage to a brightness. When the TFT turns off, the capacitor's stored voltage causes the pixel to maintain its state (brightness).

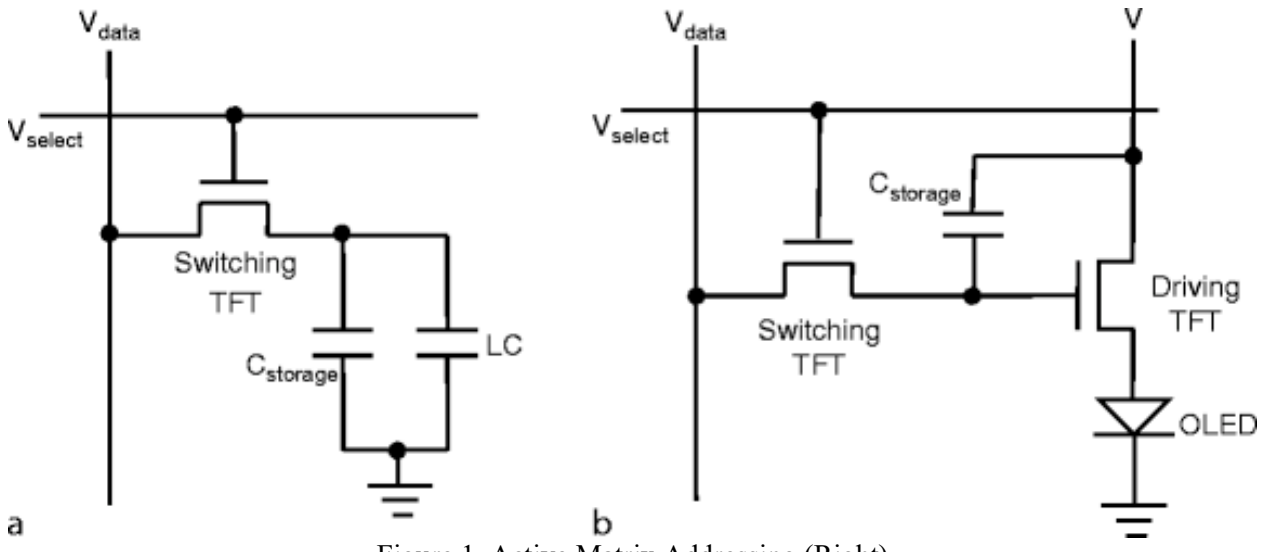



Figure 1. Active Matrix Addressing (Right)

## Procedure

The 32 by 32 RGB matrix (referred to simply as 'the matrix' from hereon) utilizes RGB LEDs as pixels (see Theory section). In this configuration, there are a total of $32 \times 32 = 1024$ RGB LEDs, or 3072 total LEDs, since there are 3 channels per LED. The matrix makes no distinction between a red, green, or blue LED.

In order to change the brightness of a singular LED efficiently, a variable constant current circuit is utilized (see Figure 1). This circuit utilizes feedback in order to keep a constant current flowing across the load. The amount of current that the load is held at can be modified by changing the voltage applied on the gate of the drive transistor. Since LED brightness is approximately proportional to the forward current through the LED, the brightness can be predictably influenced through a change in voltage.

The constant current circuit utilizes a sense resistor ($R_{sense}$) and a feedback transistor in order to keep current constant. As the control voltage increases, the current flowing through the drive transistor also increases. As the current increases, the voltage drop across $R_{sense}$ increases as well. Once it reaches a certain threshold, the feedback transistor is activated, providing a discharge for the base pin of the drive transistor. This prevents the current from further increasing or decreasing.

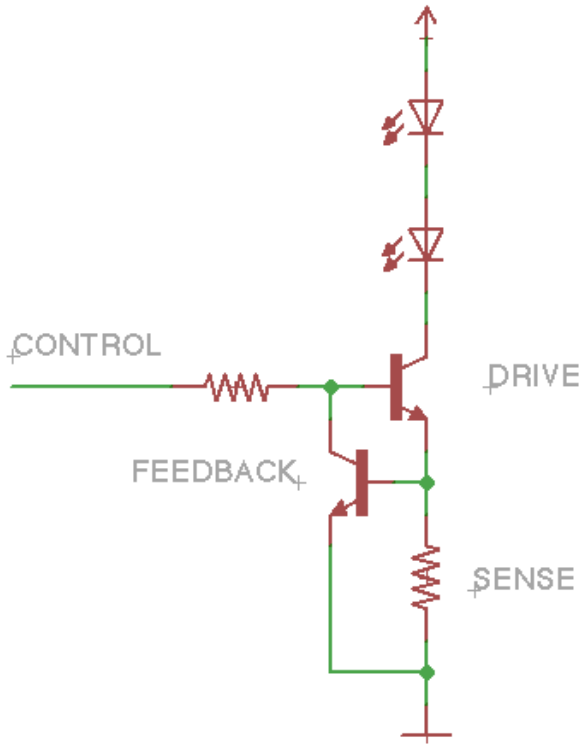



Figure 1. Variable Constant Current LED Driver

The constant current LED driver can be controlled using any variable voltage, including PWM. The matrix makes use of an R/2R ladder, because it allows a binary value to be easily converted to a voltage.

A demultiplexer, also known as a decoder, is a circuit that allows one signal line to be forwarded to multiple signal lines (see Figure 2).

Figure 2 depicts a 1:4 demultiplexer. In this configuration, there is 1 input, and 4 outputs. Depending on the select lines, 1 of the 4 outputs is active. For example, when both select lines are low, output A is selected, meaning that F is connected directly to A. The inactive lines are set to a high-impedance state, preventing them from interfering with external signals.

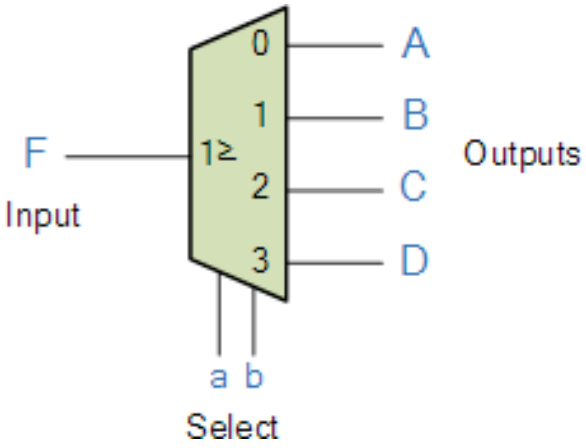



Figure 2. 1:4 Demultiplexer

A large demultiplexer can be built from smaller demultiplexers, using multiple stages. In a configuration with *n* stages, the first $n - 1$ stages are used as decoders and are connected to the enable pins of the next stage (see Figure 3). The stages closer to the left are the more significant bits, as they move the selected output further than those close. Each stage has common select pins, since only 1 chip per stage is active at any given time.

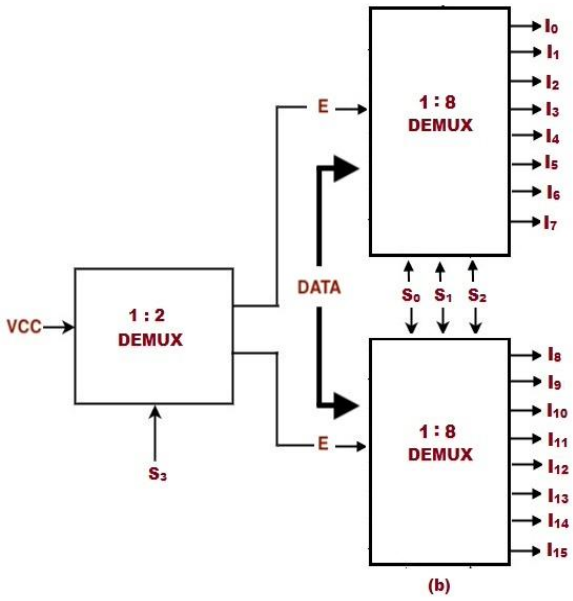



Figure 3. 2-Stage Demultiplexer

The matrix utilizes a 1:3072 demultiplexer, which is built from 1:8 demultiplexers in 4 stages. The first stage is 1:6. The next stage is 1:48, and is built by putting a 1:8 multiplexer on each of the 6 outputs from the first stage, as $6 \times 8 = 48$. The third stage is 1:384, which similarly multiplies the previous stage by 8. The final stage demultiplexes each of the 384 outputs into an additional 8, for a total of $384 \times 8 = 3072$.

The matrix utilizes this large multiplexer in order to drive each pixel without relying on a microcontroller. In this configuration, the 1:3072 demultiplexer (Q0-Q3071) is used, allowing each LED to have its own pin.

The input of the demux is connected to a variable constant current circuit. Each output is connected to an LED in a logical manner (see Figure 4).

**Figure 4. Demultiplexer Output Map**

| Output | LED | Position |
|---|---|---|
| Q0 | Red | Top left |
| Q1 | Green | Top left |
| Q2 | Blue | Top left |
| Q3 | Red | Top second-left |
| … | N/A | N/A |
| Q3071 | Blue | Bottom right |

With the LEDs connected sequentially in a row-then-column matrix configuration, each LED is effectively assigned a number. LED 0 is on Q0, LED 1 is on Q1, up to LED 3071 which is on Q3071. In this configuration, a 12-bit binary counter is utilized to scan through the LEDs one at a time. A 12-bit counter has a maximum value of $2^{12} - 1 = 4095$. Since there are only 3072 LEDs in use, additional control logic is implemented in the counter.

When the counter reaches 3072, it must reset to 0. When 3072 is expressed as a 12-bit binary value, the result is 110000000000. The only time that both S11 and S10 are high is when it outputs 3072. As a result, NAND logic is implemented. When S11 and S10 both go high, the output of a NAND gate, which is fed into the active-low clear input of the SN74HC163DR counter, goes low. Additionally, a 20 MHz crystal is attached to the clock input. This causes the counter to constantly count from 0-3071 at a rate of $\frac{20{,}000{,}000}{3072} = 6510$ Hz.

The demultiplexer configuration automatically shows the image stored in a RAM chip on the matrix. The frame data (brightness values for each pixel) are stored in RAM in an identical fashion to the layout of the matrix: left-to-right, top-to-bottom. This means that in address 0, the brightness value for the LED on Q0 is stored, in address 1, the value for the LED on Q1 is stored, up to address 3071 which stores the brightness value for the LED on Q3071.

As a result, the output of the counter also goes to the address pins of the RAM chip. This guarantees that the address accessed in RAM, and therefore the value expressed on the RAM output, will correctly be matched to the LED selected by the demultiplexer.

The outputs of the RAM chip go directly to the aforementioned R/2R ladder. This directly converts the stored brightness value into a corresponding current, and therefore brightness of the LED. Using this design, as the counter continuously cycles from 0-3072, each pixel displays the correct colour.

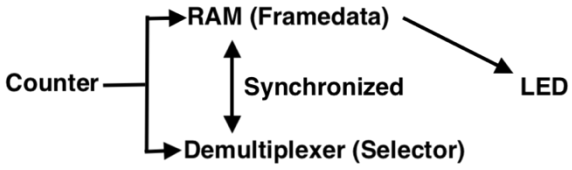



Figure 5. The Matrix Operational Flowchart

The matrix includes 2 RAM chips that act as a *double buffer*. A double buffer is a system in which one buffer is being read from while one is being written to (see Figure 6). This prevents flickering, and allows each to be fully accessed by an entire system at a time.

To accomplish this, multiplexers are utilized. The data pins of each RAM chip are multiplexed between the R/2R ladder and PORTD of an ATmega328P microcontroller. The address pins are multiplexed between two different 12-bit counters. The first 12-bit counter has its clock controlled by a 20 MHz crystal, while the second has its clock controlled by the ATmega328P. Finally, the select pins of the two multiplexers are inverted, such that it can be controlled by a singular I/O pin.

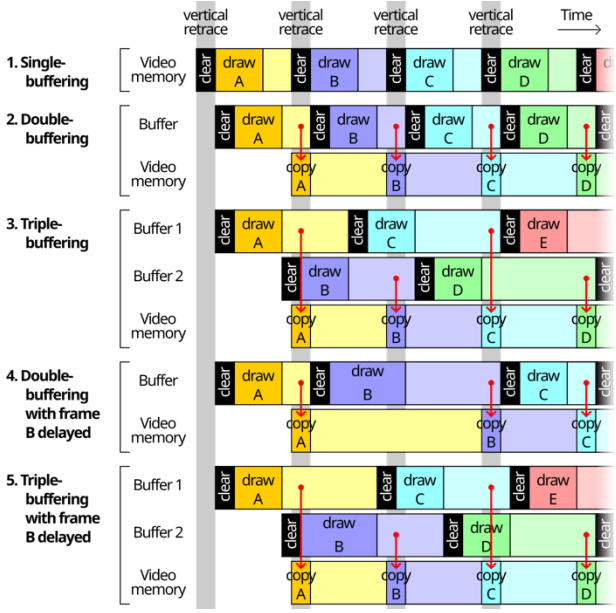
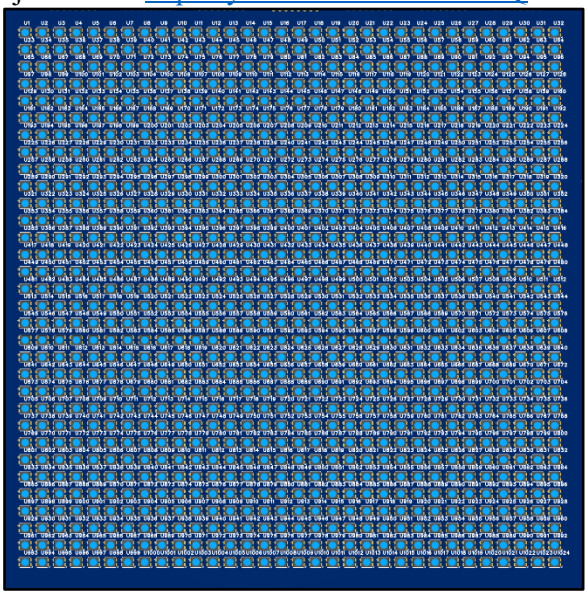



Figure 6. Multi Buffering

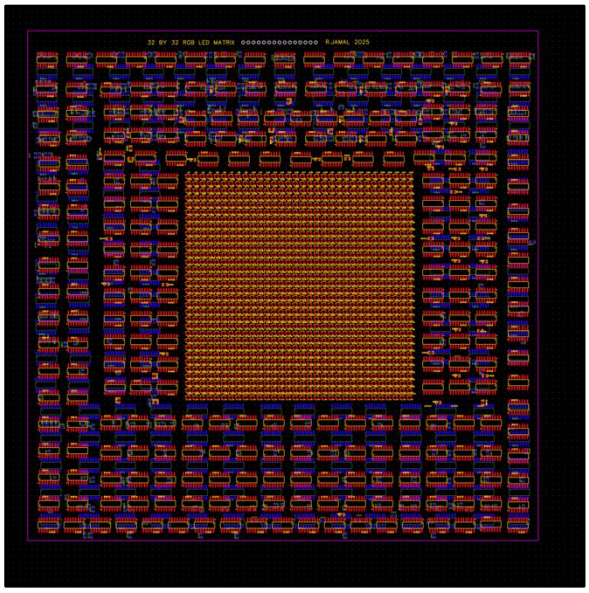
The ATmega328P is connected to a OV7670 camera module. A timer1 interrupt runs at a rate of 60Hz. Each time the interrupt is triggered, the RAM chip is switched, and the current frame from the OV7670 is captured and stored to the inactive RAM chip. This results in a 60 Hz video feed from the camera being displayed on the matrix. While the actual video feed is only 60 Hz, the true refresh rate of the display is the aforementioned 6510 Hz.

## Media

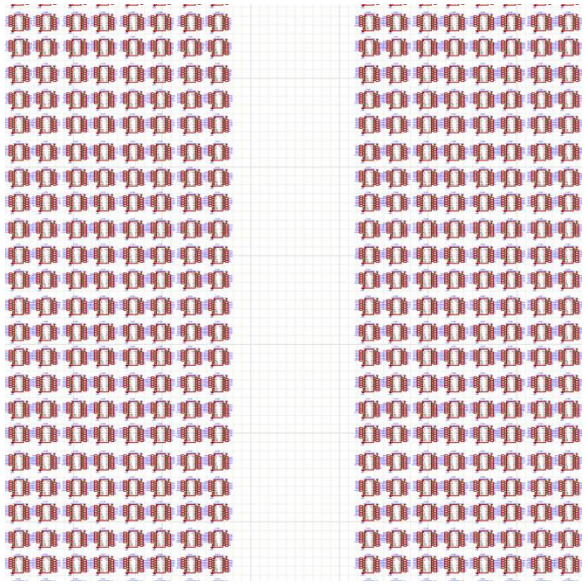
Project video: https://youtu.be/PLDhlLMfOPQ



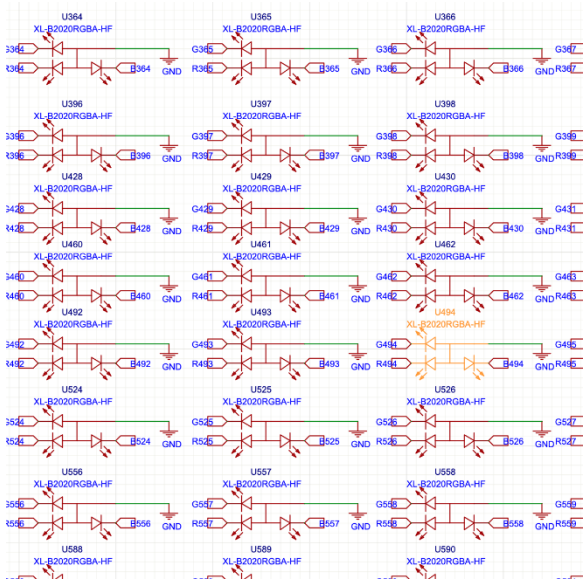
Matrix 3D Model/Rendering



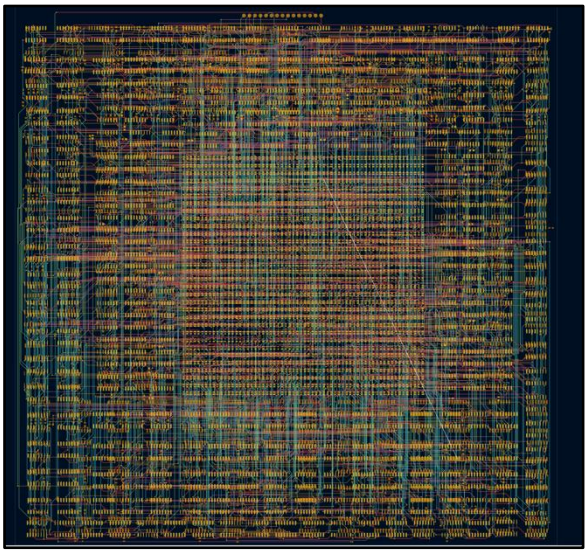
PCB (Top and Bottom) EasyEDA Pro View



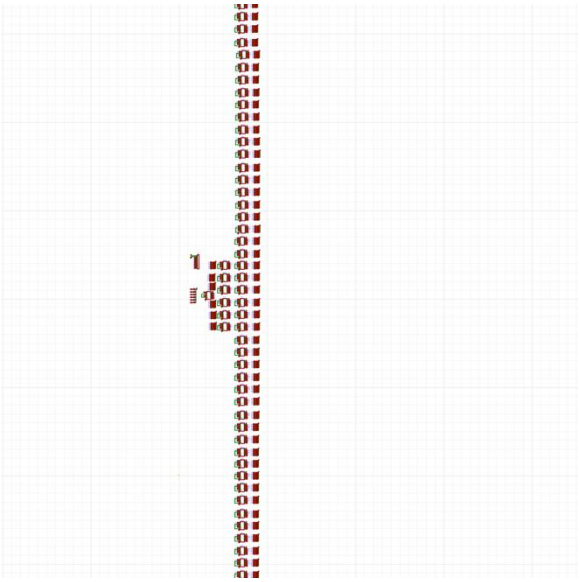
Demux Stage 4 Excerpt



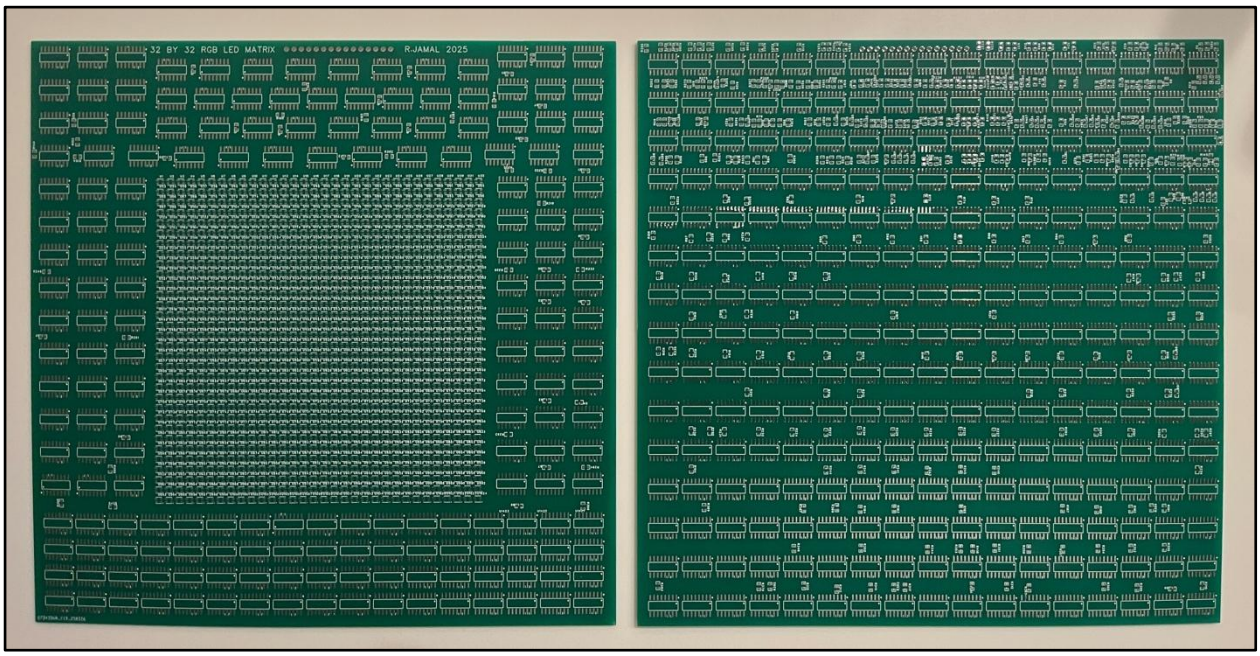

LEDs Schematic



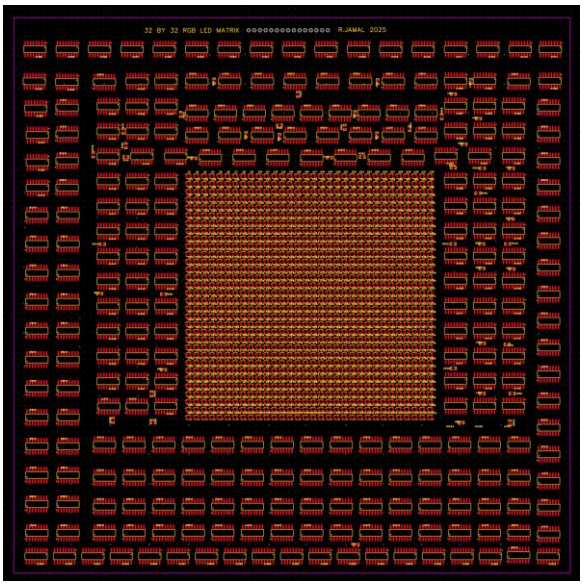
PCB Routing Progress



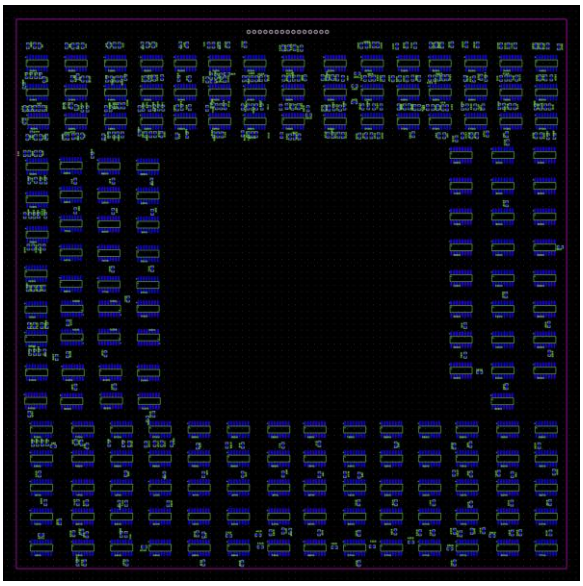
Demultiplexer Deocoding Circuit




Unrouted PCB (V1) Top Side (Left) and Bottom Side (Right)



PCB (Top) EasyEDA Pro View (V2)



PCB (Bottom) EasyEDA Pro View (V2)

## Reflection

This has been quite an ISP. Like usual, it was crazy rush up until the week before the presentation, at which point it died down because I had already done everything I could; unlike past ISPs, however, my project did not actually work. I could chalk this up to any number of issues, but the biggest (and perhaps most significant) two are ambition and stubbornness. This was, without a doubt, by far the most ambitious project I have undertaken in this course. The sheer number of PCB connections should have been enough to scare me off at the beginning, but it didn't.

My somewhat stubborn nature is often what leads me to success: I don't back down from a challenge, which usually allows me to prevail and finish successfully. Unfortunately, when paired with my imagination for what this project could have been, it made that extremely difficult to fulfill. While the issue I faced was 'only' the ludicrous routing time, it should not be a surprise that there was an unforeseen issue.

What I would take away from this project is that you can't always get what you want. Sometimes you have to be able to make compromises on the specifics in order to finish within a realistic timeframe. That being said, this certainly is not the end of this project. I have (ironically) spent much more time on this ISP than any in the past, and I believe that I have come to far to give up now. Especially with the removal of the Short ISP, I have to finish; while I have learned a lot about PCB design, demultiplexers, displays, and display drivers, this is not the note that I want to end my ISP career on. I have been thinking long and hard, and I have already come up with a slightly modified approach that could solve my routing issues. Additionally, I am not far off from a finished product. Once my PCB is routed, the project is just about finished, and I have already double and triple checked the schematic, so I am fairly certain that it will work. Overall, this has been a great project, though it is certainly not the end (Short ISP or not).

# Project 3.7 Double Dabble

## Purpose

The purpose of the Double Dabble project is to utilize the shift and add 3 (double dabble) algorithm to convert a 12-bit reading from a potentiometer to a 5-nibble packed BCD value, and echo it using ASCII equivalents through the USART interface.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/Tasks.html#DoubleD

Project GitHub: https://github.com/rohan-development/3.7-DoubleDabble

## Theory

There are many different *sets* of data. A set refers to a type of data with some number of elements. If it is a closed set, there are a finite number of elements. If it is an open set, there are an infinite number of elements. While elements of a set, can be represented by characters, when written on paper, a computer cannot represent any type of data aside from binary numbers. All stored data in a computer is either a 0, or a 1. The most common way to interpret a string of 0s and 1s is as a binary number; this is not the only interpretation, however. As long as the computer (or sender and receiver) agree on what set is represented by some number of binary digits, the data can represent any set. For example, if it is agreed that 5-bit value represents the lowercase letter, 00000 could be said to represent a, and 11001 could be said to represent z. This idea gives rise to the American Standard Code for Information Interchange (ASCII). ASCII is utilized to represent characters using a value. An ASCII value is a byte long, although it only uses 7 of the 8 bits. There are 128 possible characters, 95 of which are printing characters, and the remaining 33 are control characters. When an ASCII character is sent or received, only the raw value is sent. Values are chosen carefully: capital and lowercase letters simply differ by 1 bit (see Figure 1).

**Figure 1. ASCII Table**

| Character | ASCII (Bin) | ASCII (Dec) | ASCII (Hex) |
|---|---|---|---|
| 0 | 0011 0000 | 48 | 0x30 |
| 1 | 0011 0001 | 49 | 0x31 |
| 2 | 0011 0010 | 50 | 0x32 |
| 3 | 0011 0011 | 51 | 0x33 |
| … | | | |
| 7 | 0011 0111 | 55 | 0x37 |
| 8 | 0011 1000 | 56 | 0x38 |
| 9 | 0011 1001 | 57 | 0x39 |
| A | 0100 0001 | 65 | 0x41 |
| B | 0100 0010 | 66 | 0x42 |
| C | 0100 0011 | 67 | 0x43 |
| … | | | |
| Z | 0101 1010 | 90 | 0x5A |
| a | 0110 0001 | 97 | 0x61 |
| b | 0110 0010 | 98 | 0x62 |
| c | 0110 0011 | 99 | 0x63 |
| … | | | |
| z | 0111 1010 | 122 | 0x7A |

## Procedure

In order to display anything on the Arduino serial monitor, ASCII must be utilized (see Theory section). In a high-level language, when the user makes a call to the serial monitor, the Serial library converts everything to an ASCII value behind the scenes, before sending the ASCII code over USART. The serial, monitor automatically interprets a number to be its ASCII equivalent. For example, if decimal 51 is received, 51 is not displayed; '3' is displayed because its ASCII value is decimal 51. Therefore, in order to send a number to the serial monitor, it must be first converted to a stream of several ASCII characters.

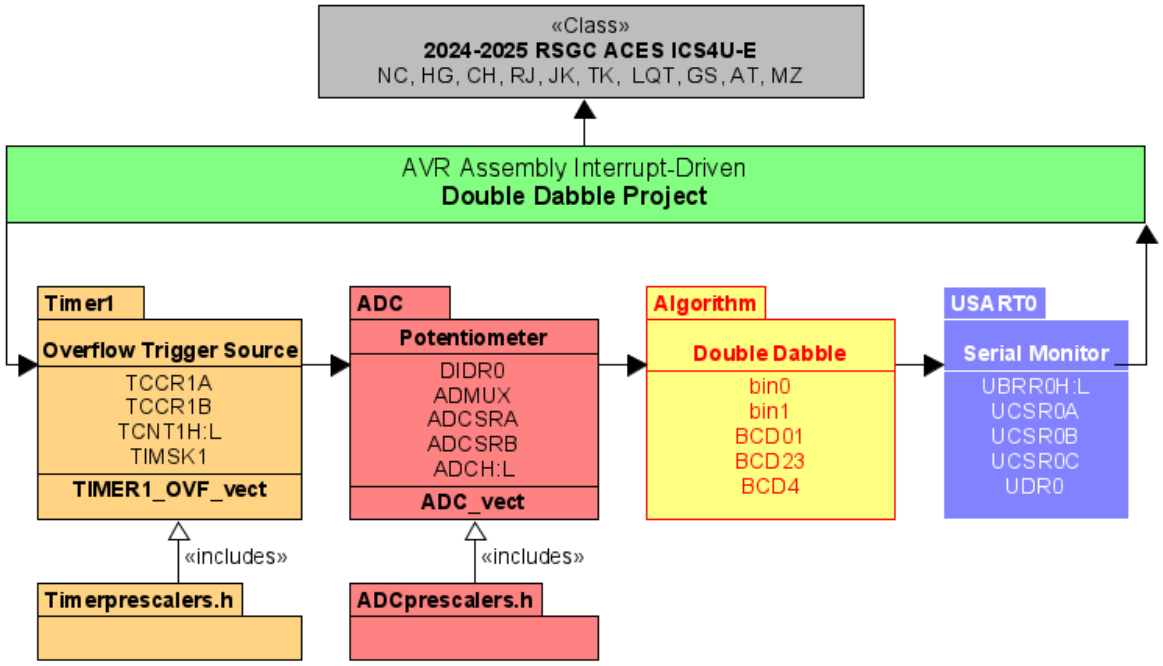



Figure 1. Double Dabble Flowchart

Figure 1 depicts the way the double dabble project prints the value read by the ADC unit on the serial monitor, from a high-level perspective. First, Timer1 is utilized to generate an interrupt upon it overflowing. When the interrupt is generated, the ADC begins a reading, and the Timer 1 overflow vector is called. This ISR simply resets the value in Timer1 to zero. When the ADC reading is completed, the 12-bit value is stored in the ADCL and ADCH register pair and a second interrupt is triggered. This interrupt loads the ADCL and ADCH registers into r18 (bin0) and r19 (bin1), respectively. Next the number that is now split between bin0 and bin1 is separated into its ASCII decimal digits, and sent to the serial monitor via USART.

The bulk of the computation done within this project is the separation into its distinct digits. In a high-level language, the modulo and division operator could be utilized in the same fashion as performed in Project 2.5.3. In assembly language, the modulo operator does not exist. Instead, the binary number must be separated into decimal digits. One candidate to perform this function is binary-coded decimal (BCD) (see Project 1.4:E). BCD essentially represents a number using 4 bits to represent each decimal digit of a number. For example, the binary number 1101 is equivalent to 13 in decimal. As such, its BCD conversion is 0001 0011. By converting the ADC reading to BCD, the reading can be easily converted to a stream of ASCII characters.

In order to convert a standard binary number to BCD, the double dabble, or shift-and-add-3 algorithm is used. This is an extremely simple algorithm that can be used to convert any binary number to a BCD value. The binary number is first set up with appropriate "scratch space" that can hold the converted BCD number (which will be larger). Since each digit will take up a nibble (4 bytes) the scratch space is separated into nibble-sized chunks (see Figure 2). First, the binary input is shifted left, spilling into the BCD scratch space. This is repeated until the value of any 1 nibble is equal to or exceeds the value 5. In this case, 3 is added to that nibble, and the system is shifted again. This runs until the original binary input has been shift *n* times, where n is the bit length of the input.

**Figure 2. Double Dabble Sample Conversion**

| BCD2 | BCD1 | BCD0 | Input | Action |
|---|---|---|---|---|
| 0000 | 0000 | 0000 | 10010111 | N/A |
| 0000 | 0000 | 0001 | 0010111. | Shift |
| 0000 | 0000 | 0010 | 010111.. | Shift |
| 0000 | 0000 | 0100 | 10111… | Shift |
| 0000 | 0000 | 1001 | 0111…. | Shift |
| 0000 | 0000 | 1100 | 0111…. | Add |
| 0000 | 0001 | 1000 | 111….. | Shift |
| 0000 | 0001 | 1011 | 111….. | Add |
| 0000 | 0011 | 01111 | 11…… | Shift |
| 0000 | 0011 | 1010 | 11…… | Add |
| 0000 | 0111 | 0101 | 1……. | Shift |
| 0000 | 1010 | 1000 | 1……. | Add |
| 0000 | 0101 | 0001 | …….. | Shift |

While the algorithm is extremely simple to use and understand, the way in which it works is slightly more complex. The most important part of the double dabble algorithm to understand is that shifting left is equivalent to multiplying by 2. Since the valid values of a nibble in BCD are 0-9, when the number is greater than or equal to 5, multiplying by two will result in an invalid nibble. The action of adding 3 effectively handles the carrying of the number. For example, if a nibble has a value of 7, shifting, or multiplying by 2 would yield a 14, which is not valid in BCD. It would go from 0111 to 1110. When 3 is added to 7, 10 is obtained. 10 in binary is 1010. When this value is shifted left, the leftmost 1 overflows and the remaining value is 0100, or 4, which forms the units digit of the aforementioned 14. Thus adding 3 can be interpreted as the carry action of the double dabble algorithm.

Before the MCU is ready to perform ADC readings and make calls to the USART peripheral, the peripherals must be setup. For this, the main code makes calls to functions called `TIMER1Setup`, `ADCSetup`, and `init_USART`. These functions simply initialize their respective peripherals, in the same way that `peripheral.begin()` would do in a high-level language. The `Timer1Setup` function performs the following features: firstly, it loads a 0 into the TCCR1A register, which configures the timer in normal mode. Then, it sets the prescaler to 64 by placing predefined T1ps64 into register TCCR1B. This is to trigger the ADC readings approximately twice per second. Finally, the timer is cleared, and enabled by setting the TOIE1 bit in the TIMSK1 register.

The next peripheral that is utilized in this project is the ADC. The ATmega328P includes a SAR ADC, similar to the one built in Project 3.5, although the ATmega328P includes a 12-bit one, instead of a 7-bit one like Project 3.5 does. The ADC converts analog signals and voltages to discrete digital values (see Figure 3).

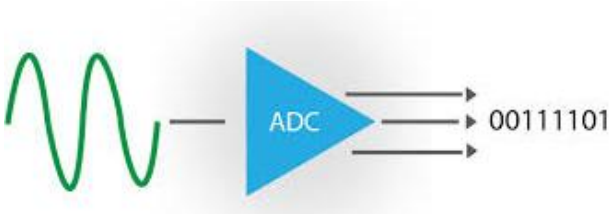



Figure 3. The Matrix Operational Flowchart

The ATmega328P ADC can operate in both in free-running mode, or interrupt-triggered mode. In free-running mode, when the conversion is completed, it automatically runs the next conversion. This means that the ADC is constantly running. While the processor can be doing other things at the same time, it is not ideal from a power consumption standpoint. As depicted in Figure 4, the peripheral never stops running. For the vast majority of applications, the better mode is interrupt-driven mode. This allows the ADC to run upon the overflow of Timer1. In this configuration, when Timer1 overflows, the ADC automatically starts a conversion.

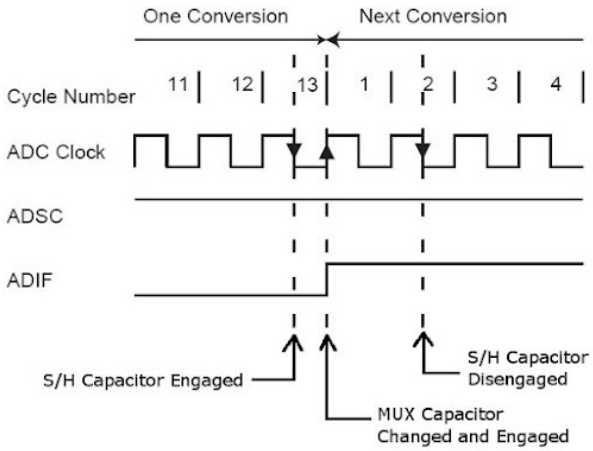



Figure 4. ADC Free-Running Mode

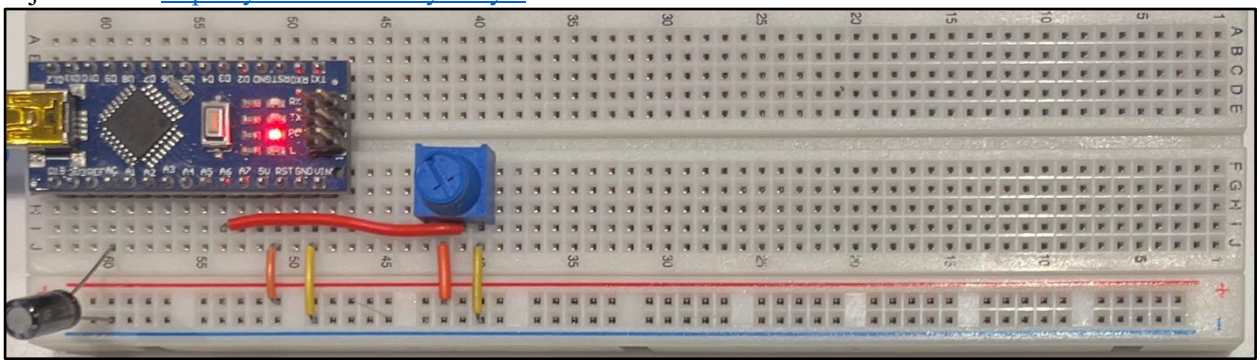
| Figure 5. ADC Modes | | | | |
|---|---|---|---|---|
| **ADTS3** | **ADTS2** | **ADTS1** | **ADTS0** | **Trigger Source** |
| 0 | 0 | 0 | 0 | Free-Running Mode |
| 0 | 0 | 0 | 1 | Analog Comparator |
| 0 | 0 | 1 | 0 | External Interrupt Request 0 |
| 0 | 0 | 1 | 1 | Timer/Counter0 Compare Match |
| 0 | 1 | 0 | 0 | Timer/Counter0 Overflow |
| 0 | 1 | 0 | 1 | Timer/Counter1 Compare Match B |
| 0 | 1 | 1 | 0 | Timer/Counter1 Overflow |
| 0 | 1 | 1 | 1 | Timer/Counter1 Capture Event |
| 1 | 0 | 0 | 0 | Timer/Counter4 Overflow |

One major advantage of using assembly language over the standard high-level alternatives is that the processor can perform a second task while the ADC is running a conversion. In a high-level Arduino C, the `analogRead()` function is a blocking function, meaning that the program stops while it is executed. In assembly, the ADC can be started, and the processor can perform another task until the ADC complete interrupt is reached. In this case of the double dabble project, no additional action is performed.

As soon as the ADC completes its conversion, the ADC complete ISR is called. This simply calls the double dabble algorithm, which converts the ADC reading to BCD. Finally, 48 is added to each nibble, or BCD digit, upgrading it to a full byte in ASCII format. 48 is added because it is the offset to get the raw numbers into ASCII-formatted numbers. These bytes are then placed in the UDR0 register, which automatically transmits it over the USART.

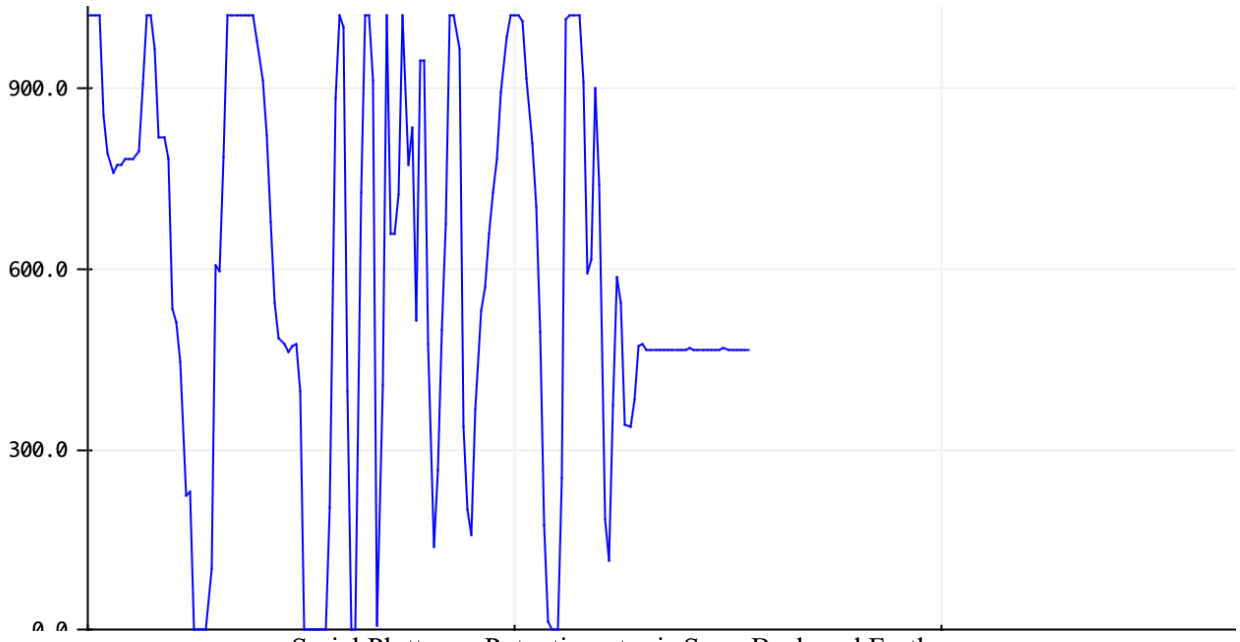
## Media

Project video: https://youtu.be/BewBydGeyIs



Double Dabble Project


Serial Monitor as Potentiometer is Spun Back and Forth




Serial Plotter as Potentiometer is Spun Back and Forth

## Reflection

This project has taught me several things (perhaps unsurprisingly). Firstly, it taught me how much easier high-level code is than assembly. I knew it was easier, but I have a huge appreciation for high-level code now. Just knowing that my entire 200-line assembly code could have been replaced by 2-3 high-level instructions is both humbling and fascinating. Also, it showed me the somewhat niche use-cases of assembly language. I have heard Mr. D'Arcy talk about the many advantages several times, but this is the first time I am really witnessing and understanding the huge savings. For some reason, I find it cool that you can have the processor doing something while waiting for an ADC approximation to complete. This really does make a lot of sense, but the `analogRead` function always implied that you couldn't, and that the CPU was busy during conversion.

It also gave me a bit of a refresher on time management. I completed the code for the project well before the deadline, so I was ahead of most people in the class in that element. That made me think that I was further ahead than I really was, and writing the report took a lot more time than I had anticipated, leading to a late submission. This was an excellent reminder that projects should be slow burns, rather than several quick spurs. Regardless, this was an excellent project, although I am somewhat sad that assembly is out of the curriculum in both the ACES course, and likely next year in university.

# Project 3.8 4517: Gray Code

## Purpose

In addition to introducing gray code, Boolean algebra, and Karnaugh maps, the purpose of the 4517 gray code project is to create a singular 4 input to 7 output decoder in which the first 16 gray code values are mapped to their respective hexadecimal values in a seven segment display-friendly format, using digital logic gates.

## Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/Tasks.html#4517G

## Theory

Combinational logic is the backbone on which computer systems are built. Having a predictable set of outputs from a given set of inputs is the foundation upon which computers operate. While creating a truth table from a given set of logic gates is a relatively straightforward process, the opposite process (which arguably more important) is a seemingly-impossible task. Implementing the truth table using the least number of gates possible is a challenging task. As such, a simple method to produce a Boolean expression (from which a set of logic gates can be derived) exists. This is known as *Karnaugh Mapping*. A Karnaugh map consist of the inputs arranged in a truth table-like grid (see Figure 1). The difference lies in the fact that the outputs are ordered in *gray code*. Gray code is an alternative binary encoding in which adjacent values only differ by 1 bit. In a 2-bit system, values are taken from adjacent vertices of a square in a cartesian coordinate system.

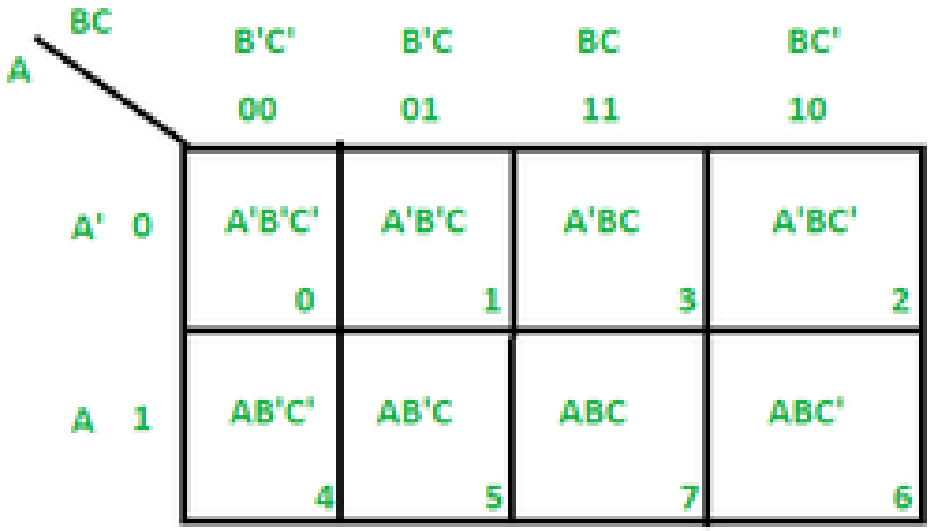



Figure 1. Sample Karnaugh Map

To derive the Boolean expression from the Karnaugh map, each square is listed with the Boolean product (AND) of the variables on the top and left. Then, rectangles of 1s are found in powers of 2 (i.e., 1, 2, 4, 8 1s). In Karnaugh mapping, the edges are considered to wrap around. For example, in Figure 1, square 2 and 0 would be considered adjacent squares. Finally, the Boolean expression is derived by considering any variables that are constant in all squares. The process is repeated until each 1 is part of at least 1 group. The derived Boolean terms are all ORed together, leading to a sum of products. The process can also be performed by finding groups of zeros, provided that the final output is inverted.

## Procedure

In order to convert a gray code value to its respective hexadecimal value on a seven-segment display, combinational logic is utilized. Four gray code inputs are converted to 7 outputs, 1 per display segment. In this configuration, the truth table for each segment is created, and then the required logic gates are synthesized from the truth table. For this, Karnaugh Mapping (K-Mapping) is utilized (see Theory section). To ensure the least number of logic gates is utilized, the maximum number of shared terms is utilized.

Figure 1 depicts the K-Map for segment a (see Figure 2). As shown in the truth table, segment a is only inactive when the character being shown is 1, 4, b, or d. From this map, an expression must be derived. Since a larger group will have less variables in common, they are more desirable as fewer variables aligns with fewer required logic gates. Hence, groups of 8 are most desired, then 4s, then 2s, with 1s least desired.

In Figure 1, there are no groups of 8, so groups of 4 are the most desired. The first group of 4 is the first column, $\bar{B}\bar{A}$, which is added to the expression. The next group of 4 is the rectangle in the middle. Theoretically, a group of 4 could be created from the 4 corners, however it is not necessary because subsequent segments will require the same groups of 2 anyways.

Finally, there are 3 groups of 2. Hence, the final expression is the sum of each created product:

$$a = \bar{A}\bar{B} + AC + \bar{B}\bar{C}D + \bar{A}B\bar{C} + AB\bar{D}$$

**Figure 1. A Segment K-Map**

| a | $\bar{B}\bar{A}$ | $\bar{B}A$ | $BA$ | $B\bar{A}$ |
|---|---|---|---|---|
| $\bar{D}\bar{C}$ | 1 | 0 | 1 | 1 |
| $\bar{D}C$ | 1 | 1 | 1 | 0 |
| $DC$ | 1 | 1 | 1 | 0 |
| $D\bar{C}$ | 1 | 1 | 0 | 1 |

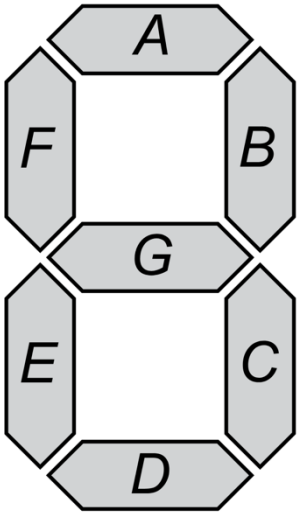



Figure 2. Seven Segment Display

This process is repeated for each segment, repeating as many terms as possible to reduce the number of AND gates utilized. Segment b and segment c both utilize their respective inverse K-Map, in which groups of zeros are found and then the final outputs are inverted (see Figures 3 and 4). The benefits of this method are especially apparent in segment b as it requires only 3 terms, each of which are also used in other segments. Without this method, at least 4 terms would be required, some of which are unique to segment b. This reduces the required number of gates, even when accounting for the final inverter.

**Figure 3. B Segment K-Map**

| a | $\bar{B}\bar{A}$ | $\bar{B}A$ | $BA$ | $B\bar{A}$ |
|---|---|---|---|---|
| $\bar{D}\bar{C}$ | 1 | 1 | 1 | 1 |
| $\bar{D}C$ | 1 | 0 | 0 | 1 |
| $DC$ | 1 | 1 | 1 | 0 |
| $D\bar{C}$ | 0 | 0 | 1 | 0 |

**Figure 4. C Segment K-Map**

| a | $\bar{B}\bar{A}$ | $\bar{B}A$ | $BA$ | $B\bar{A}$ |
|---|---|---|---|---|
| $\bar{D}\bar{C}$ | 1 | 1 | 0 | 1 |
| $\bar{D}C$ | 1 | 1 | 1 | 1 |
| $DC$ | 1 | 1 | 1 | 1 |
| $D\bar{C}$ | 0 | 0 | 1 | 0 |

Once the Boolean expression is derived, it is built using AND, OR, and NOT gates. The derived Boolean expression for each segment is listed in Figure 5. Each expression is derived from K-Mapping, in a fashion that allows the entire system to use as few gates as possible. This is done by sharing the maximum number of terms. In total, the system uses 24 AND gates, 24 OR gates, and 6 NOT gates.

**Figure 5. Segment Boolean Expressions**

| Segment | Expression |
|---|---|
| a | $\bar{A}\bar{B} + AC + \bar{B}\bar{C}D + \bar{A}B\bar{C} + AB\bar{D}$ |
| b | $\overline{(\bar{B}\bar{C}D + AC\bar{D} + \bar{A}BD)}$ |
| c | $\overline{(\bar{B}\bar{C}D + \bar{A}\bar{C}D + AB\bar{C}\bar{D})}$ |
| d | $\bar{A}\bar{C}\bar{D} + AB\bar{D} + AC\bar{D} + A\bar{C}D + \bar{A}CD + \bar{A}BD$ |
| e | $\bar{A}\bar{B}\bar{C} + \bar{A}CD + \bar{B}\bar{C}D + BD + AB\bar{C} + A\bar{B}C\bar{D}$ |
| f | $\bar{B}D + BC + \bar{A}\bar{B}\bar{C} + \bar{A}\bar{C}D + AC\bar{D}$ |
| g | $\bar{B}D + AB + AC + BC + \bar{A}B\bar{D}$ |

**Figure 6. Build Scheme**

| AND Number | AND Expression | OR Number | OR Expression | NOT Expression |
|---|---|---|---|---|
| **A1** | A'B' | 1 (START A) | A1\|A2 | A |
| **A2** | AC | 2 | 1\|A4 | B |
| **A3** | B'D | 3 | 2\|A6 | C |
| **A4** | A3&C' | 4 | 3\|A8 | D |
| **A5** | A'C' | 5 (START B) | A9\|A10 | b |
| **A6** | A5&B | 6 | 5\|A4 | c |
| **A7** | AB | 7 (START C) | A4\|A11 | |
| **A8** | A7&D' | 8 | 7\|A12 | |
| **A9** | A2&D' | 9 (START D) | 5\|A13 | |
| **A10** | A16&B | 10 | 9\|A8 | |
| **A11** | A5&D | 11 | 10\|A15 | |
| **A12** | A8&C' | 12 | 11\|A17 | |
| **A13** | A5&D' | 13 (START E) | A18\|A17 | |
| **A14** | C'D | 14 | 13\|A4 | |
| **A15** | A14&A | 15 | 14\|A19 | |
| **A16** | A'D | 16 | 15\|A20 | |
| **A17** | A16&C | 17 | 16\|A21 | |
| **A18** | A1&C' | 18 (START F) | A3\|A22 | |
| **A19** | BD | 19 | 18\|A18 | |
| **A20** | A7&C' | 20 | 19\|A11 | |
| **A21** | A9&B' | 21 | 20\|A9 | |
| **A22** | BC | 22 (START G) | 18\|A7 | |
| **A23** | A'B | 23 | 22\|A2 | |
| **A24** | A23&D' | 24 | 23\|A24 | |

Figure 6 depicts the scheme in which the circuit is built, including what gates are connected and shared. This scheme was selected to reduce the total number of logic gates as much as possible.

The final important feature of the 4517: Gray Code Decoder is the self-counting feature. Functionally, this feature makes the project not unlike Project 1.4. A Counting Circuit. Similar to Project 1.4, the device counts up starting at zero; this, however, is where the similarities end. Firstly, the Gray Code Decoder counts up to 0x0F, completing a full nibble, instead of a single BCD packet. Additionally, it counts in gray code, not standard binary. This makes the counting procedure relatively complex.

To count in gray code, the CD4516 is utilized in conjunction with a binary to gray code converter (see Figure 7). This effectively creates a gray code counter. The converter is implemented using XOR gates

By XORing each of the three low binary bits with its adjacent bit, the low 3 bytes of gray code are formed. Since the high byte is the same in both gray code and binary, no manipulation is required.

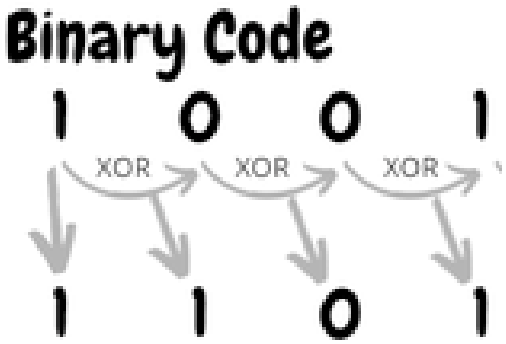


Figure 7. Binary to Gray Code Conversion

To switch from manual counting from the absolute gray code encoder to automatic, clock-pulse powered counting, the following procedure is performed: firstly, the position of the encoder is set to zero. The encoder works using pulldown resistors, which means that when the encoders signals are all low, the lines are released. Next, a DIP rocker bank of 4 switches is set to connected the gray code counter to decoder.

Figure 8 depicts the gray code sequence. The unique feature that distinguishes gray code is that only one bit changes between adjacent numbers.

One interesting quirk of this, when paired with the aforementioned gray code counting system employed in this project, is that when the high bit is disabled (i.e., the high switch remains off), the device will count up to 7, pause for a cycle as it shows 8, since 8 only differs by the high bit. Finally, the device will count down to 0 from 7.

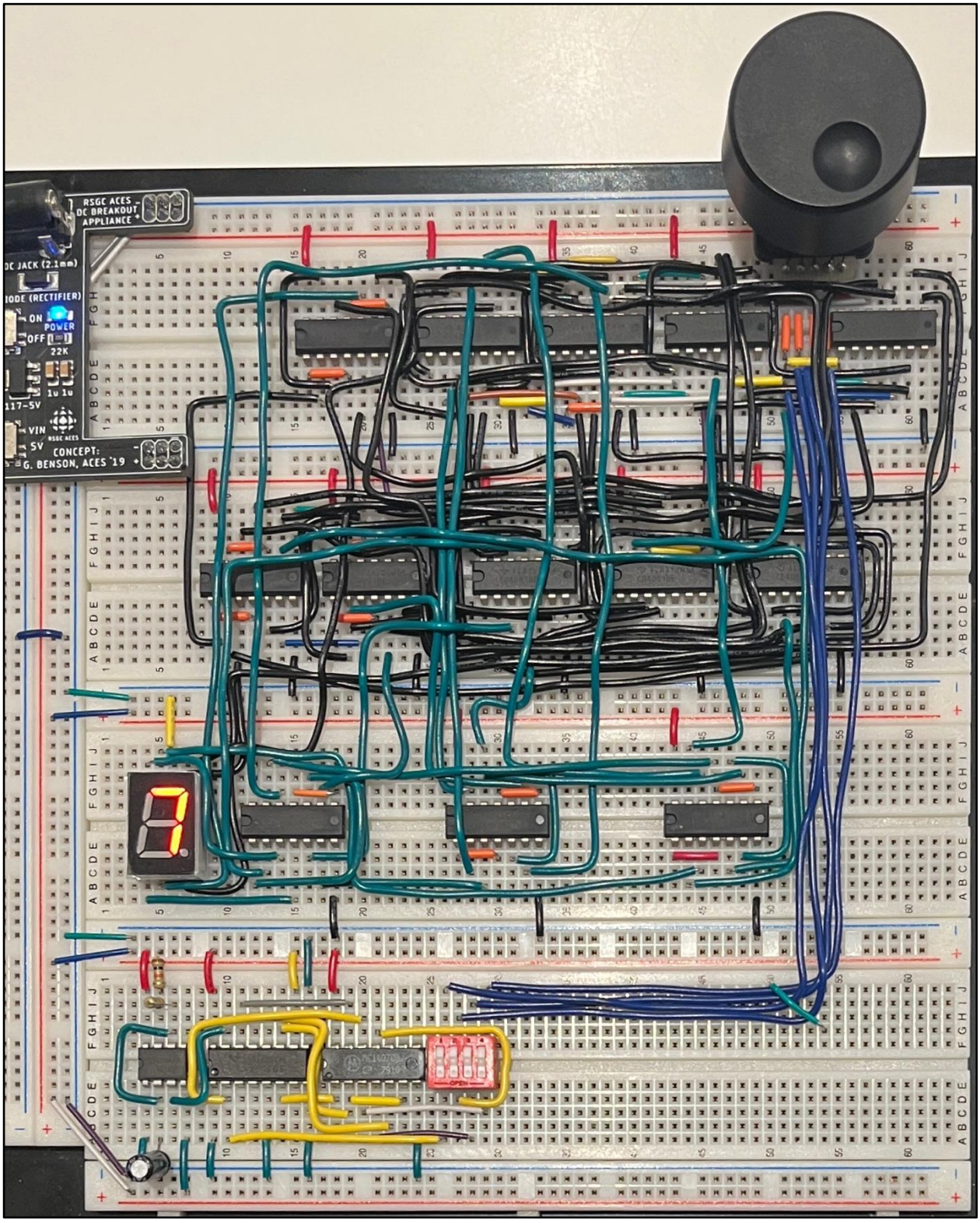
This behaviour is explained by the following property of gray code: the low three bits are symmetrical about the number 7; when the high bit is ignored, counting down from 15 is equivalent to counting up from 0, until the number 7 is reached. This is why the device will count up to 7, and then down again to 0.

**Figure 8. Gray Code vs Binary vs Decimal**

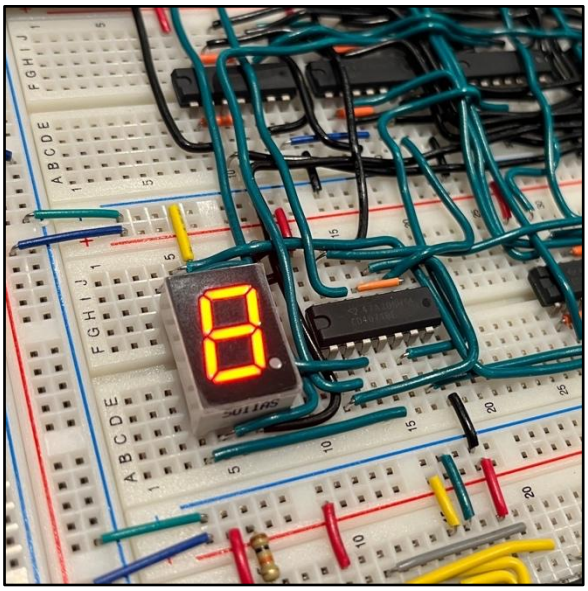
| Decimal | Binary | Gray Code |
|---|---|---|
| 0 | 0000 | 0000 |
| 1 | 0001 | 0001 |
| 2 | 0010 | 0011 |
| 3 | 0011 | 0010 |
| 4 | 0100 | 0110 |
| 5 | 0101 | 0111 |
| 6 | 0110 | 0101 |
| 7 | 0111 | 0100 |
| 8 | 1000 | 1100 |
| 9 | 1001 | 1101 |
| 10 | 1010 | 1111 |
| 11 | 1011 | 1110 |
| 12 | 1100 | 1010 |
| 13 | 1101 | 1011 |
| 14 | 1110 | 1001 |
| 15 | 1111 | 1000 |

## Media

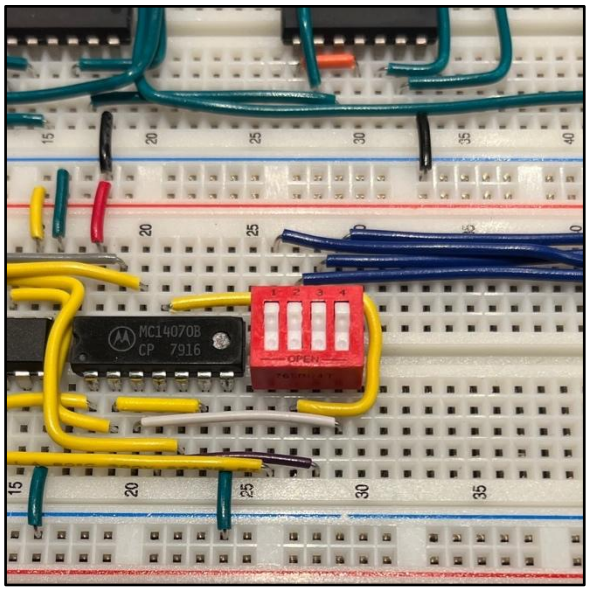
Project video: https://youtu.be/O1wqS1wA97o



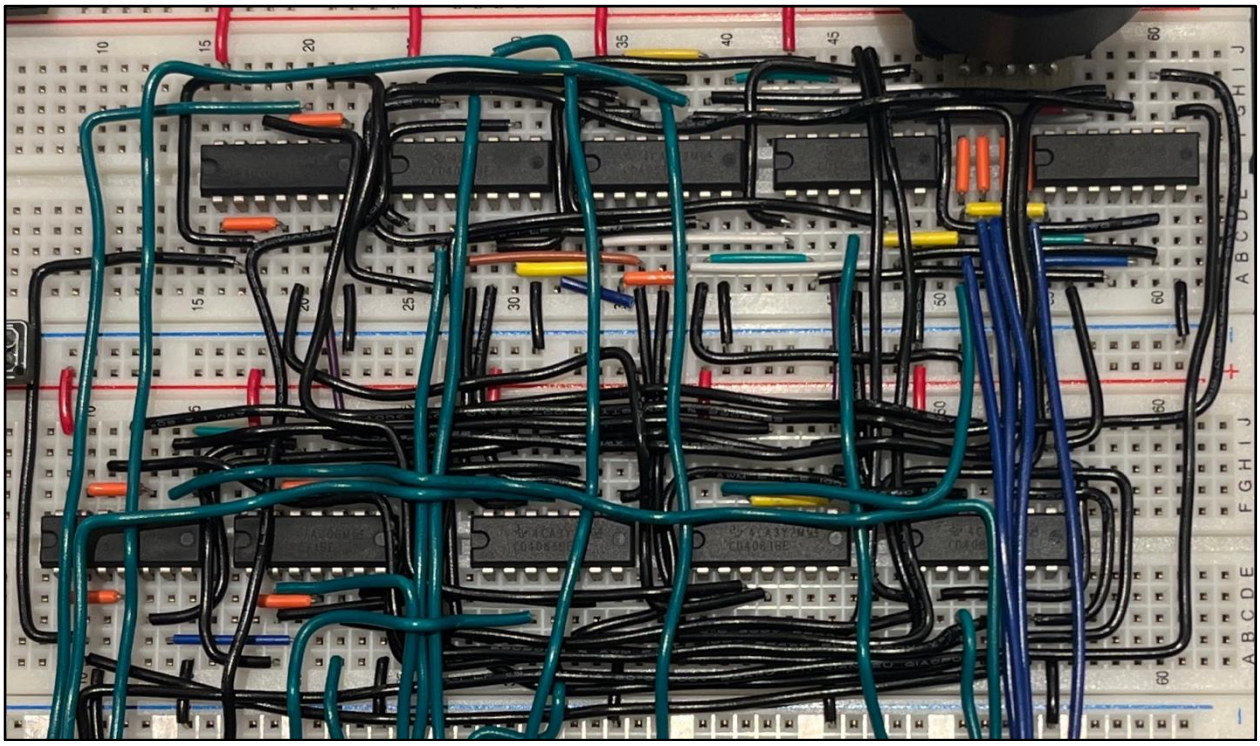

4517 Entire Circuit



Seven Segment Display (Output)




Automatic vs Manual Switch Bank



Combinational Logic (Sum of Products Implementation)

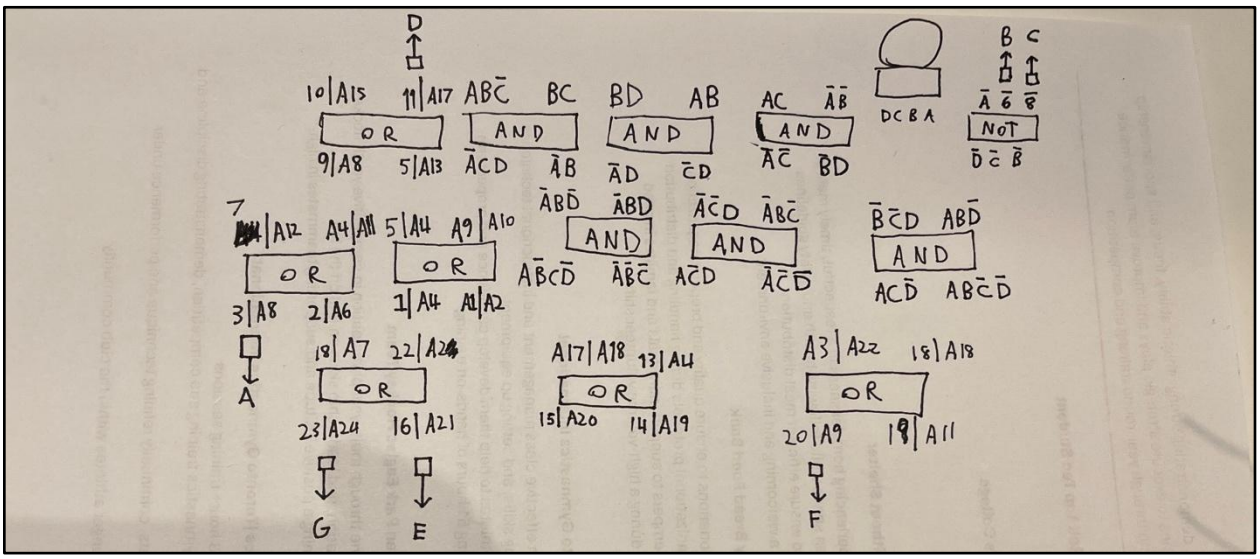
## Reflection

See Reflection.

# Reflection

As this is the last project, I felt it was fitting to give this reflection the full Heading 1 status. This is not just the reflection on the 4517 Gray Code project, it is a reflection on the first 3 years in this lifelong journey. Firstly, regarding this project:

This was a great project. I had a lot of fun building it. The satisfaction I got when I finished building was not unlike that of CHUMP. It didn't work the first time, but I luckily, I had a hunch this would be the case, so I made a map of where everything was in my circuit. This let me really quickly narrow down and isolate the problem. In fact, the debugging process took a grand total of 20 minutes to fix 4 segments. This easily could have taken hours without my map of the physical board (see below). Overall, I'm very happy with the project.




Now back to the more global reflection: I do want to point out that they are not entirely separate. In some ways, this was the perfect project to end on. I thought it was a particularly fitting/full circle moment when I got mine to count by itself. I think that this speaks to some of the core principles of the course: it was functionally almost identical to the grade 10 counting circuit. To an unseasoned observer, it could be the same thing. It counts from 0 to a number on a seven-segment display, then resets. And this, I think, is what has made these last three years so memorable. In this course, the details are everything. Having the display count in gray code instead of binary is all the difference, even if not technically visible in the end result. I think that these are skills that I will carry with me forever. Being able to make an under-the-hood distinction is something I will always do. I think that I have also learned some non-hardware-specific skills. Writing a report, time management, planning, understanding your place (grade 10:) are just some of the soft skills I've learned. I really want to say thank you so much for a great three years. This was truly a once-in-a-lifetime course, and going back to grade 10, I think it came around at the perfect time in my life. We will be in touch.