A SURVEY OF DIGITAL LOGIC CIRCUITS

Megan Smith

Project Definition

For this project, I decided to simulate several different circuits using basic analog components and an Arduino Uno. To begin, I constructed a mechanism to mimic the logic of the most elementary logic gates, including the following gates: AND, NAND, OR NOR, XOR and NXOR. Following this, I built a half adder using a quad integrated NAND circuit and a AND gate made from two NPN transistors. Next, I constructed a full adder using a hex converter integrated circuit and quad AND, OR and NAND integrated circuits. I also simulated a JK Flip Flop using a 955 timer, a quad NAND gate and two additional NAND gates built using two transistors each. I continued to use the 955 timer when putting together a simulation for a binary ripple counter using JK Flip Flops. Here I used four integrated circuits, each of which contained a JK Flip Flop circuit. Moving on from time-based circuity, I then made a 1-2 decoder/demultiplexer by using the hex converter integrated circuit and two AND gates made from two transistors each. To opposite the decoder, I next made an encoder using a quad OR integrated circuit and one OR gate made from two transistors. I concluded my project by building a multiplexer using a hex converter integrated circuit, a quad AND and a quad OR integrated circuit and by using six AND gates made from two transistors each. All of the circuits that I modelled uses blue or red LEDs to represent either the input or the output of the circuit and utilize the 5 volt power supply and ground pins supplied by the Arduino Uno.

The Basic Logic Gates

Originally I planned on beginning the project by making each basic logic gate individually using even more basic components (such as resistors and transistors), but once I realized I was going to have to build those basic gates individually for later devices anyway (such as the multiplexer circuit), I decided to put together an apparatus that will simulate each gate by using two push buttons as inputs and an LED as the output. The setup for this simulation can be seen in figure 1. Here you can see that whenever a button is pressed, a yellow LED is triggered to represent that input as having a true value and when the button is not pressed the LED remains off to represent that input as having a false value.

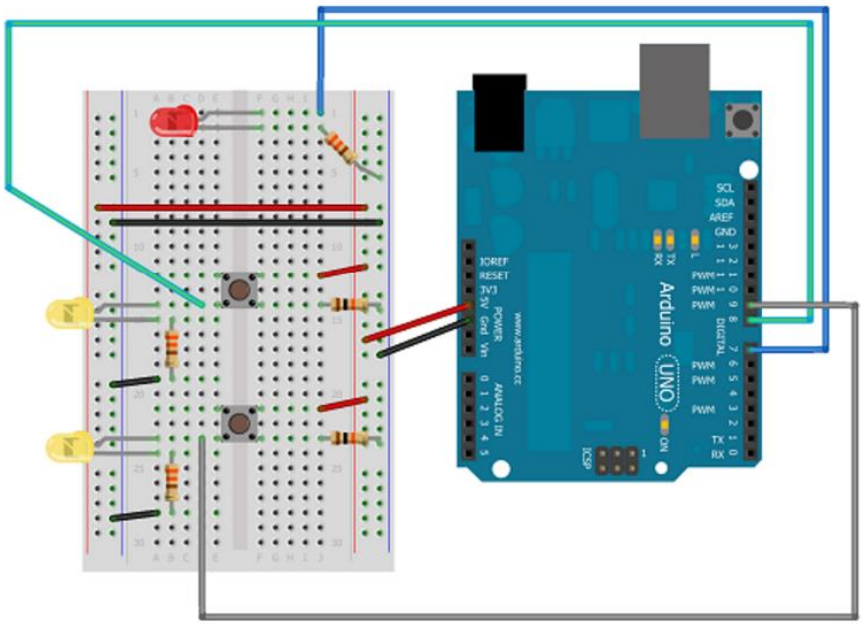


Figure 1: Basic Logic Gate Apparatus

A push button allows this true/false state to occur because the button is connected to the power supply through the Arduino (represented by red wires) and once the button is pressed, that power supply can then be accessed by the LED, which powers the LED to light up. You can also see from the diagram that each LED is connected to the ground (represented by black wires), so when power is accessible, the LED will be part of a complete circuit and will light up. The state of each input LED is connected to a pin from the Arduino board, this connection is read by the Arduino for each input LED. Based on the state of each input LED, the Arduino is programmed to output the state of the red LED based on the current code for the 'pinOutState' variable. Figure 2 shows an example of the code currently programmed onto the Arduino board. This example shows that the output LED will be true if the state of input LED A (pinA) XOR the state of input LED B (pinB) is true as well. The Arduino then takes this calculated output state for the red LED and sets the output LED to that state.

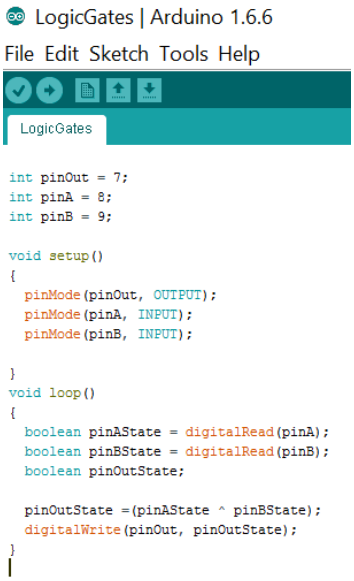
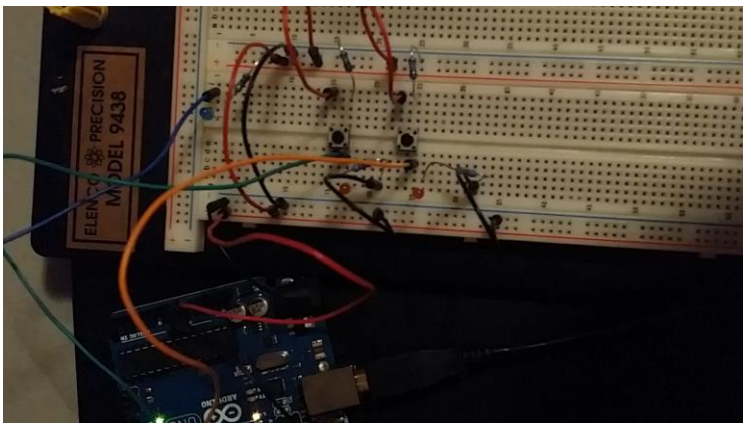
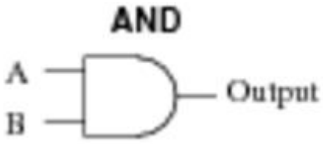


Figure 2: Arduino code for basic logic gates

The following images represent my physically implementation of the setup in figure 1. The only difference begin the two input LED begin red, and the one output LED begin blue. Each represented logic gate has an image of the initial state of the system, an icon link to a video of the working system and a truth table of the logic gate.

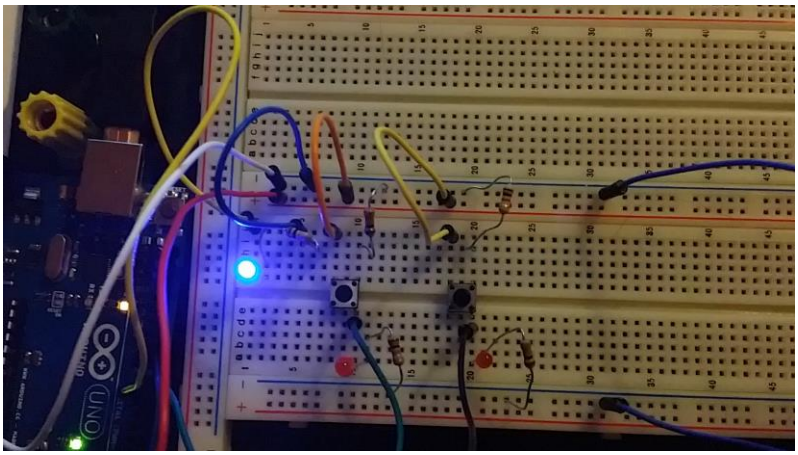
AND Gate



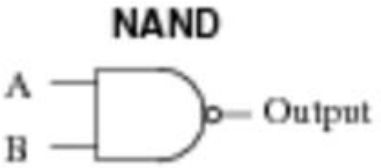


A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

NAND Gate

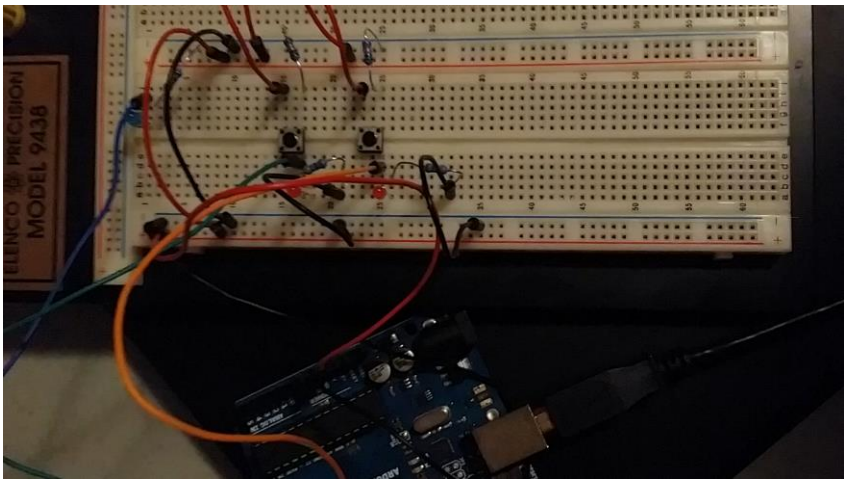


NAND_Gate.mp4

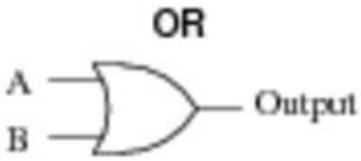


A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

OR Gate

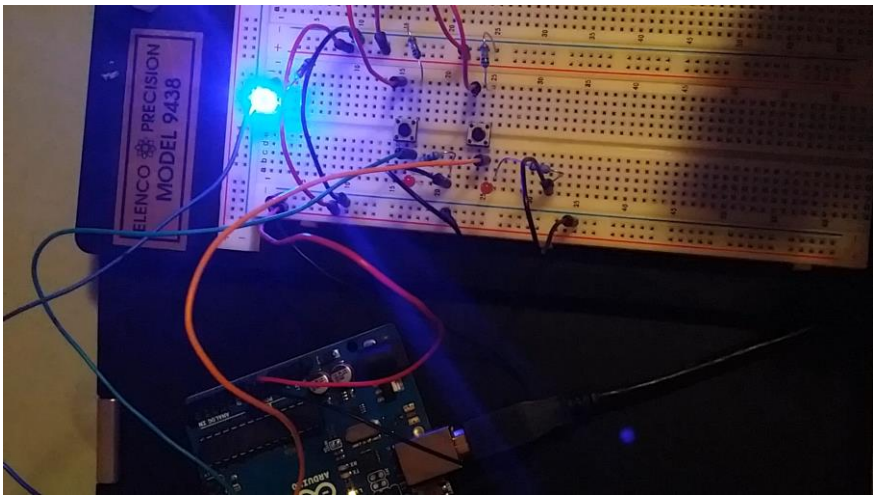


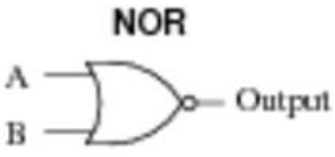
OR_Gate.mp4



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

NOR Gate

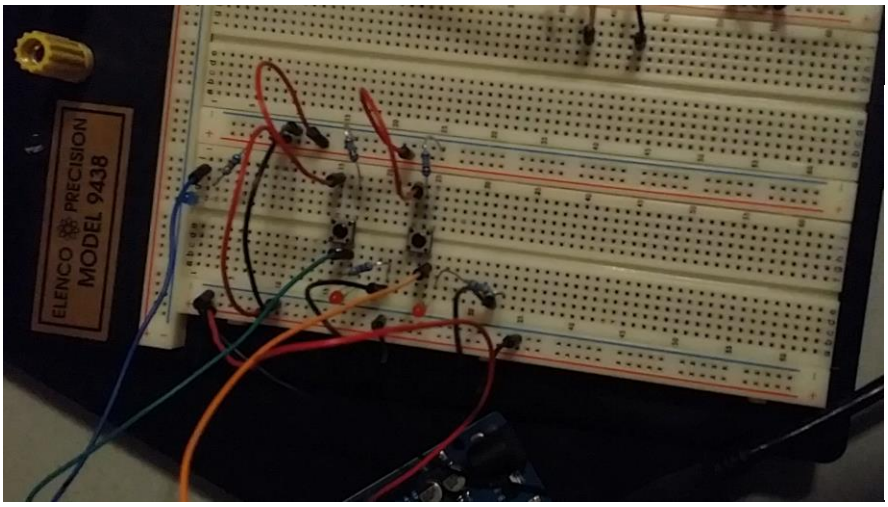


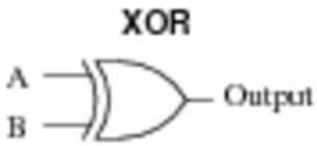


A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

NOR_Gate.mp4

XOR Gate

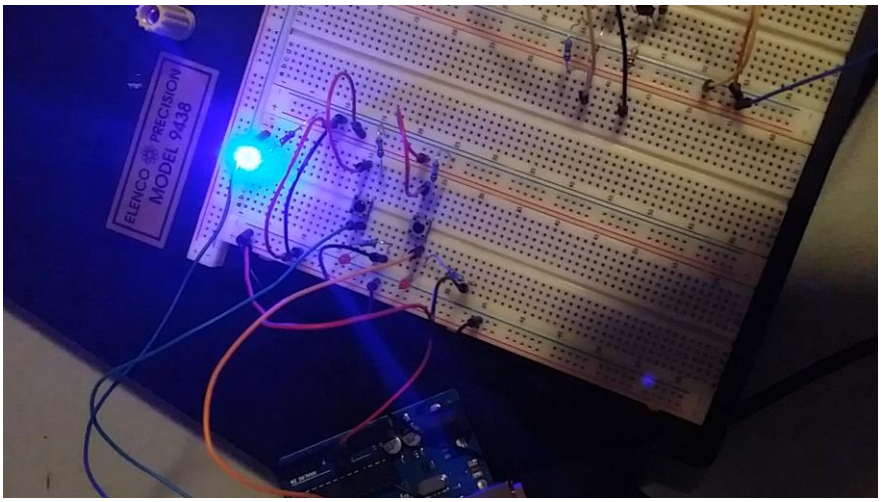


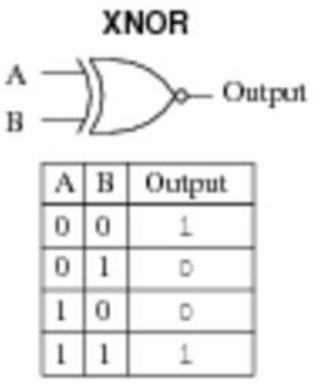


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

XOR_Gate.mp4

NXOR Gate





A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

NXOR_Gate.mp4

The Half Adder

The half adder circuit is essentially a device that performs basic addition of two binary numbers. Figure 3 visualizes this basic addition. The half adder has two inputs, X and Y, and has two outputs, S and C. S represents the actual sum of the two binary numbers and C represents the carry out of the sum.

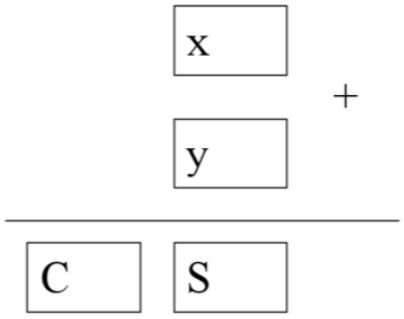


Figure 3: Addition of binary numbers performed by half adder circuit

Figure 4 shows the truth table for the half adder, the basic schematic of inputs and outputs for the circuit and the circuit diagram. If you review the truth table show, you can see if you perform binary addition on the inputs 1 and 0, the sum of the inputs is equal to 1 (which is shown in the value of S) with a carry out equal to 0 (which is shown in the value of C). This trend continues for every variations of the two inputs for the half adder. The realization of the half adder involves only two gates, the XOR and AND gates. I constructed the XOR gate by using four NAND gates as shown in figure 5. In order to make the XOR gate from four NAND gates, I used a quad NAND integrated circuit that houses four induvial NAND gates, the internal setup of which can be view in figure 6. I made the AND gate for the half adder by following the circuit diagram in figure 7, where I only needed three resistors, two NPN transistors and access to a power supply and ground (which is provided by the Arduino). From figure 7, you can see how the diagram simulates an AND gate by the positioning of the transistors in series and by the placement of the output of the circuit. The power supply is accessible only to the top transistor and the ground only to the bottom transistor. In order for a complete circuit to be made, both transistors must be saturated with power from the two inputs so the power can reach the output and then reach the ground and finally complete the circuit. This scenario happens when the AND gate has two true inputs, and then has a true output. Alternatively, if either one of the transistors had a low power state, the power supply would never reach the output and results in an incomplete circuit. This scenario has the AND gate receive a false input and has a false output.

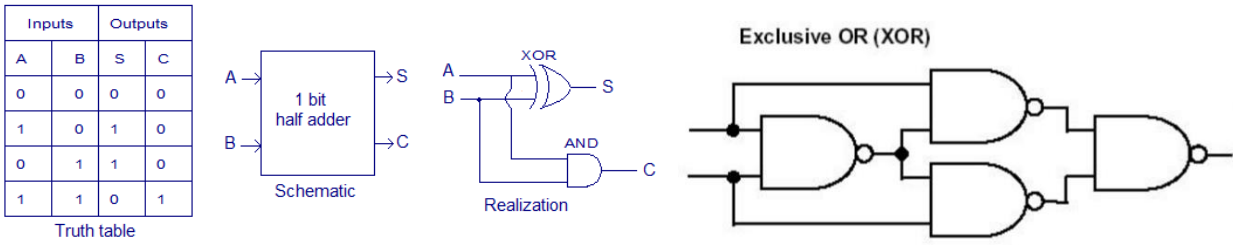


Figure 4: Half Adder truth table and circuit diagram

Figure 5: XOR gate represents as NAND gates

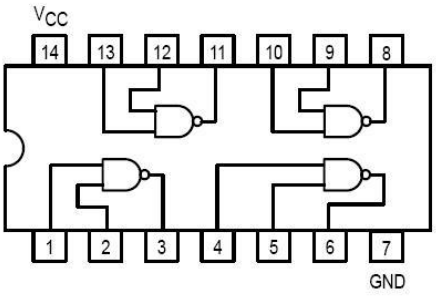


Figure 6: Quad NAND integrated circuit

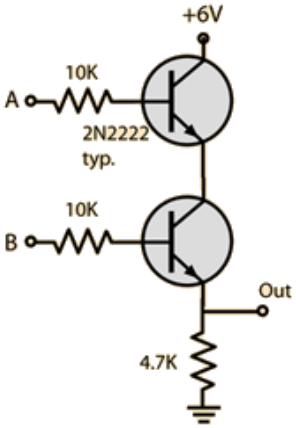
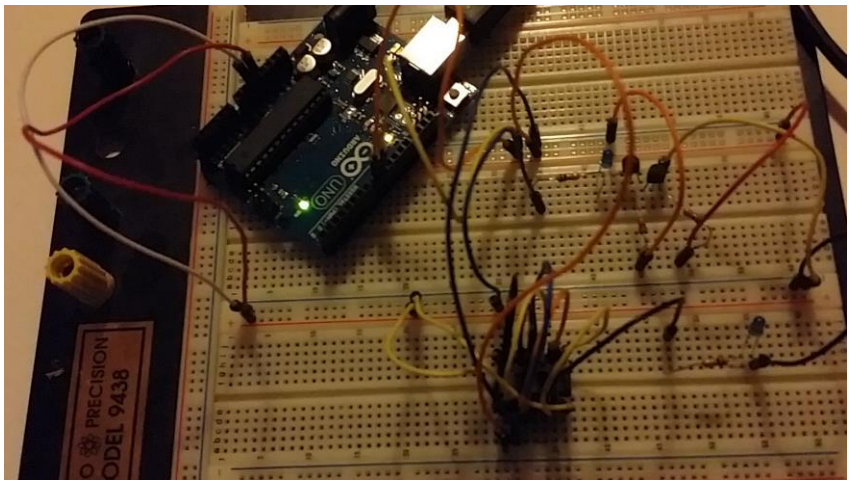


Figure 7: AND gate using two NPN transistors

Below I supplied the initial structure of the half adder circuit and a link to a video of the circuit running. I programed the Arduino to automatically change the inputs for the half adder so you can see the change of the output more fluidly. I did this by connecting the inputs of the circuit to Arduino pins and writing the state of those pins. I also added a delay method so you can see the change in the circuit, otherwise the output LED would change to fast to be noticeable. Figure 8 shows the code used. The Arduino environment has built in setup and loop methods, making it easy to define pin modes are either input or output pins and allows the system to consistently loop through the circuit, looking for changes in pin states and running the program over and over.



Half_Adder.mp4

XOR_-_HALF_ADDER

```
void setup() {
  // put your setup code here, to run once:
pinMode(8, OUTPUT);
pinMode(9, OUTPUT);

}

void loop() {
  // put your main code here, to run repeatedly:
  digitalWrite(8, HIGH);
  digitalWrite(9, HIGH);
  delay(2000);

  digitalWrite(8, LOW);
  digitalWrite(9, LOW);
  delay(2000);

  digitalWrite(8, LOW);
  digitalWrite(9, HIGH);
  delay(2000);

  digitalWrite(8, LOW);
  digitalWrite(9, LOW);
  delay(2000);

  digitalWrite(8, HIGH);
  digitalWrite(9, LOW);
  delay(2000);

  digitalWrite(8, LOW);
  digitalWrite(9, LOW);
  delay(2000);

  exit(0);

}
```

Figure 8: Arduino code for half adder

The Full Adder

The full adder is very similar to the half adder except that the full adder can perform binary addition on multiple input bits. The only thing to keep in mind is the carry out for the pervious bit is now considered the carry in input for the current bit. This can be seen in figure 9. Therefore, the full adder has a total of three inputs, two binary bits and a carry in input, and two outputs, the sum and the carry out bit. The truth table for the full adder can be analyzed in figure 10, where the addition of the binary numbers can be seen. For instance, adding input 0, 1, 1 results in an output sum of 0 and an output carry out of 1. To build the full adder circuit, I followed the circuit diagram for the full adder, shown in figure 11. Here the circuit calls for two XOR gates, two AND gates and an OR gate. I made the two XOR gates using the quad NAND

integrated circuit and the circuit diagram shown in figure 12, this XOR gate can be made using two NOT gates (used hex converter IC (integrated circuit)), two AND gates and and OR gate. Combining all the gates I need together, I used a quad AND IC since I need four AND gates and a quad OR IC since I need two OR gates. The internally workings of the quad AND and quad OR ICs is very similar to that of the quad NAND IC from figure 6.

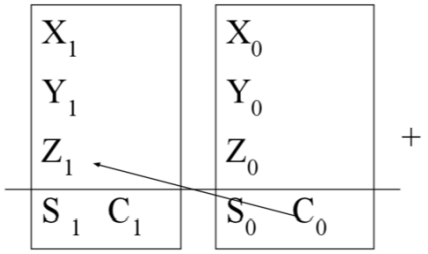
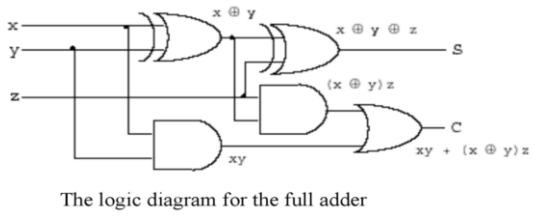


Figure 9: Addition performed by Full Adder

Input bit for number A	Input bit for number B	Carry bit input C_{IN}	Sum bit output S	Carry bit output C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 10: Truth table of Full Adder



The logic diagram for the full adder

Figure 11: Full adder circuit diagram

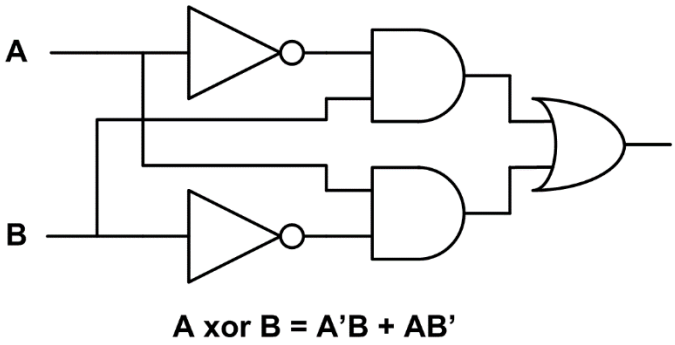
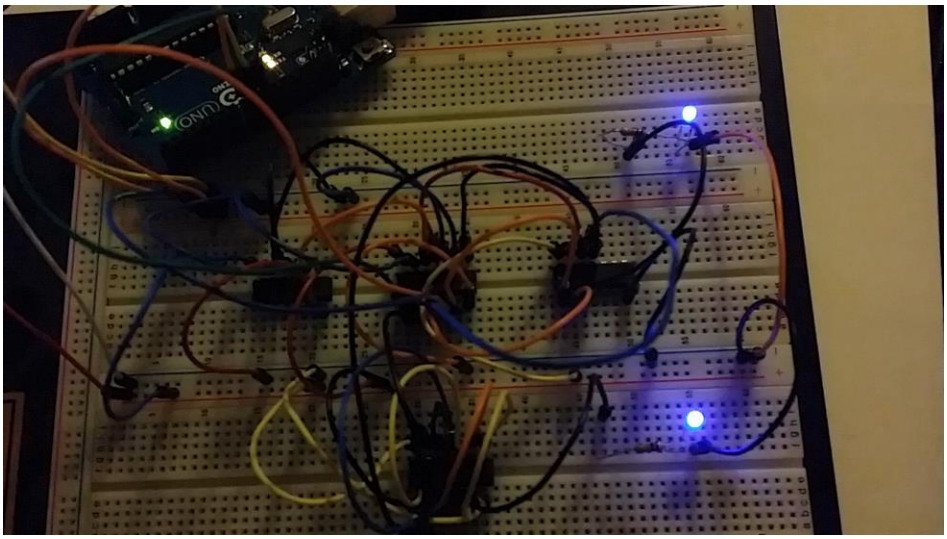


Figure 12: Circuit diagram for XOR gate

Below is an image of the initial circuit structure for the full adder and a link to the working circuity. The Arduino code used to simulate the full adder is similar to that of the half adder, where the Arduino supplies power and ground for the board, and changes the state of the full adder inputs so show the outputs more conveniently.



Full_Adder.mp4

The JK Flip Flop

The JK flip flop defines a two state device which offers basic memory for sequential logic operations. A JK flip flop is powered by the pulses of a clock. The JK flip flop has two inputs, J and K, and two outputs the state of the system and the opposite state of the system. Essentially, the outputs of the JK flip flop depend on whether or not the clock is on a high pulse or a low pulse. When the clock pulses low, the current state of the system remains, there is no change. However, the output of the system changes when the clock pulses high. Figure 13 shows how the system changes with high pulses by showing the truth table of the JK flip flop. The system state remains the same when J and K are 0, and the system toggles with the clock when J and K are 1. To build the JK flip flop are followed the circuit diagram shown in figure 14, where I used the quad NAND IC and an addition two other NAND gates designed after the circuit diagram in figure 15 using resistors and two NPN transistors. From figure 15, you can see by the positioning of the output that when both transistors are high, the power supply voltage is able to flow through both transistors and reach the ground, to form a complete circuit. When either one of

the transistors is low, the power will be diverted to the output, giving the NAND gate a true output value.

J	K	CLK	Q	$\overline{Q}$	Comment
0	0	↑	Q	$\overline{Q}$	Latch
1	0	↑	1	0	SET
0	1	↑	0	1	RESET
1	1	↑	$\overline{Q}$	Q	TOGGLE
X	X	Any Thing Else	Q	$\overline{Q}$	NO Change !

Figure 13: Truth table for JK flip flop

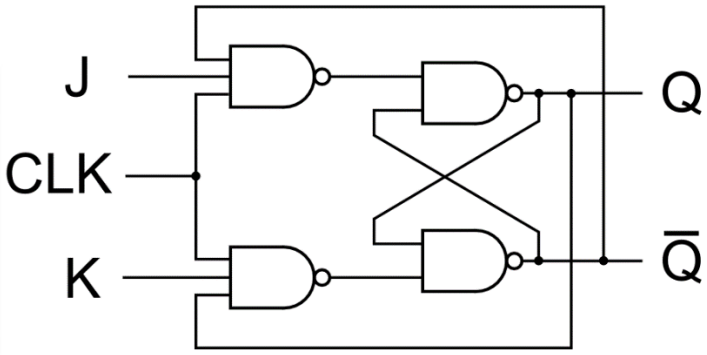


Figure 14: Circuit diagram for JK flip flop

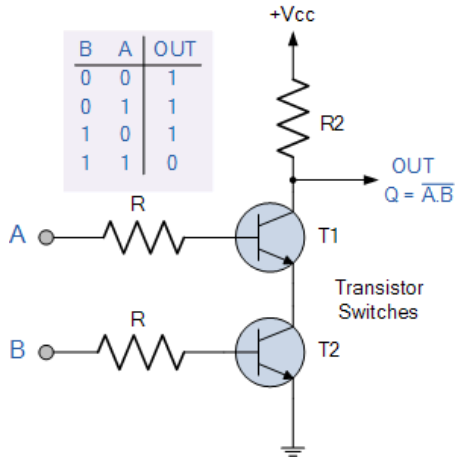


Figure 15: Circuit diagram for NAND gate using transistors

The other more important aspect of building the JK flip flop is adding the clock as one if its inputs. To connect the clock I used a 955 timer and followed the circuit diagram in figure 16. To better understand the inner workings of the 955 timer (also known as the 555 timer), refer to figure 17. The timer, like any other IC, requires a pin to connect to power and ground. The timer also has pins to begin a clock pulse (pin 2 – trigger), empty the charge stored in the capacitor that powers the clock pulse (pin 7 – discharge) and a pin to begin the pulse all over again (pin 4 – reset). The timer has other pins, but we will not discuss them for the scope of this project. The speed in which the clock pulsates is dependent on the size of the capacitor (a component which stores charge) that is connect to the trigger pin. I used the largest capacitor I had (100 micro

Farads) to slow down the pulse to view the changes in output better, but still the final version of my JK flip flop was very fast when it came time to toggle between output states. Regardless, you can see an image of the initial state of my JK flip flop circuit board, and access a video of the changing system states as well below.

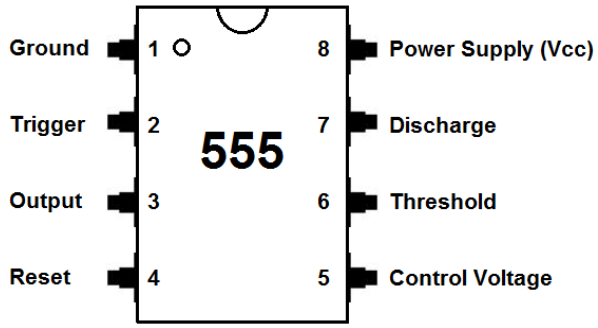


Figure 16: Schematic for 555 timer

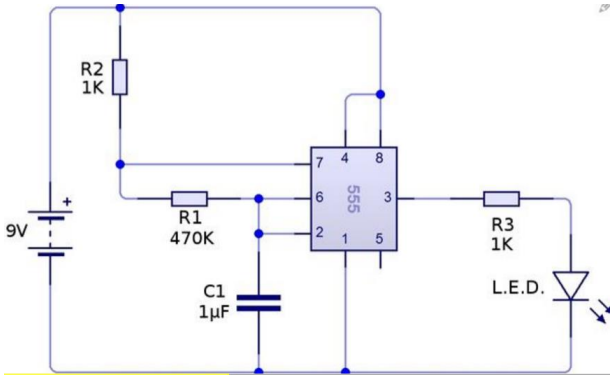
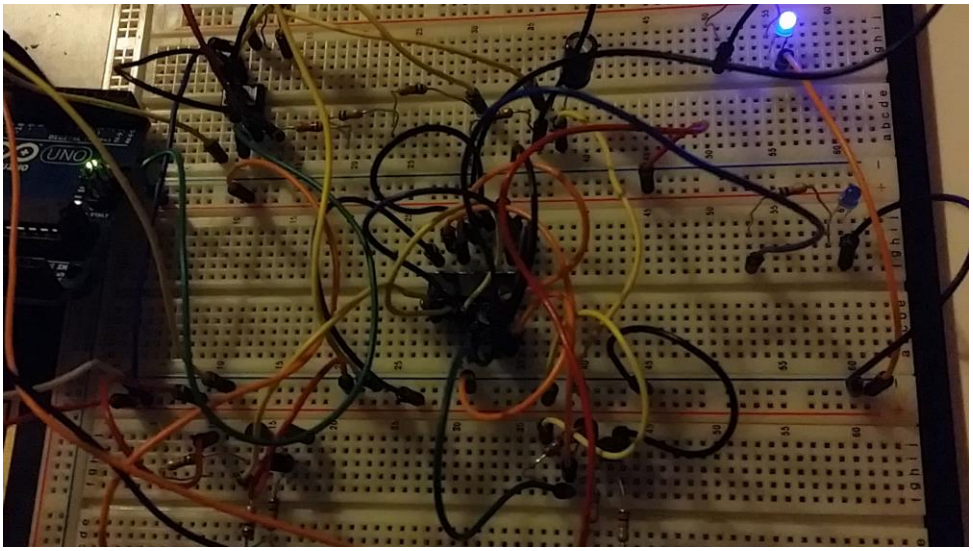


Figure 17: Circuit diagram for 555 timer



JK_Flip_Flop.mp4

The Binary Ripple Counter using JK Flip Flops

One of the main reasons I decided to previously build a JK flip flop versus a different flip flop, like the D flip flop, is because the JK flip flop can also be used as a T (toggle) flip flop. This is done by tying the values of J and K together as a single input. I used these types of JK flip flops when building a binary ripple counter. I needed four JK flip flops to construct the counter, as shown from the circuit diagram of a binary ripple counter in figure 18. From the figure, you can see that the counter requires only two inputs, a single high value for J and K, and a clock pulse supplied to the first JK flip flop. The rest of the JK flip flops in the counter have their clock inputs connected the outputs of the JK flip flop before them. The setup of clock inputs and outputs in this system is what makes the binary ripple count an asynchronous counter. The delay between the responses of successive flip flops generate a ripple effect of the final output. Remember, only the first flip flop is triggered by the clock, the remaining flip flops are triggered by the output of the proceeding flip flops. To build the counter, I used four JK flip flop ICs, each of which house two flip flops. You can see the schematic of the JK flip flop in figure 19. I had to use four of these IC instead of just two because, for some reason, the IC would only allow on JK flip flop to function properly at one time.

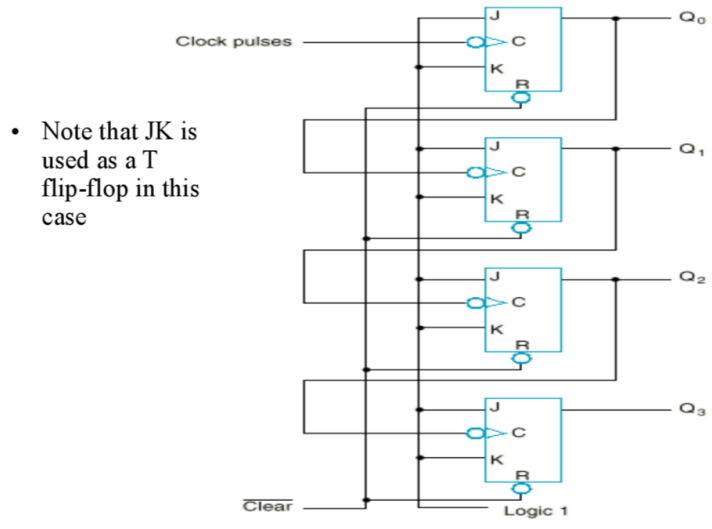


Figure 18: Circuit diagram for binary ripple counter

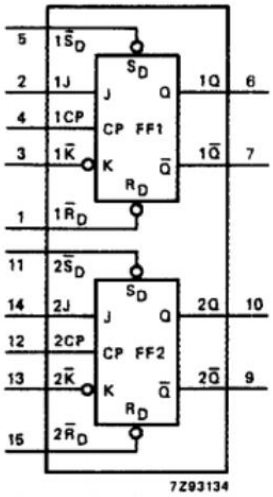


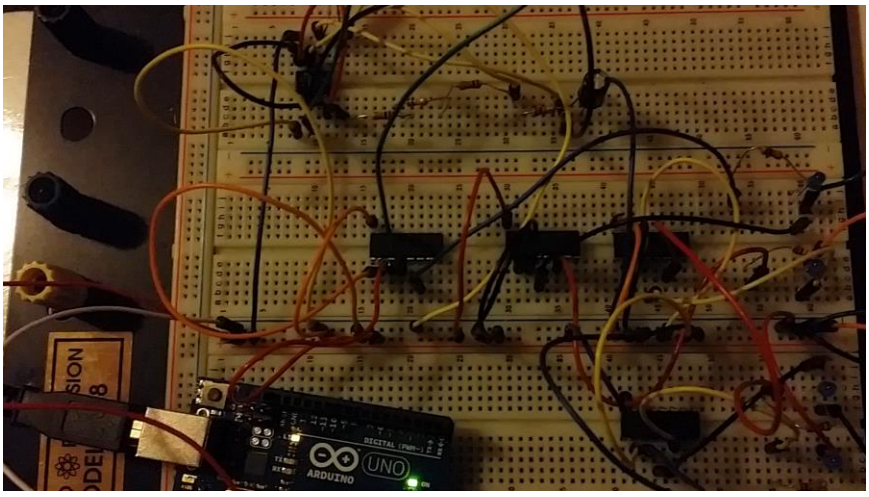
Figure 19: Schematic for JK flip flop IC

The final output of a binary ripple counter is essentially a binary counter. The counter adds one to the current binary value with each clock pulse. You can see this binary counting in the binary ripper counters truth table in figure 20.

Clock	Q3	Q2	Q1	Q0
1	0	0	0	0
2	0	0	0	1
3	0	0	1	0
4	0	0	1	1
5	0	1	0	0
6	0	1	0	1
7	0	1	1	0
8	0	1	1	1
9	1	0	0	0
10	1	0	0	1
11	1	0	1	0
12	1	0	1	1
13	1	1	0	0
14	1	1	0	1
15	1	1	1	0
16	1	1	1	1

Figure 20: Truth table for binary ripple counter

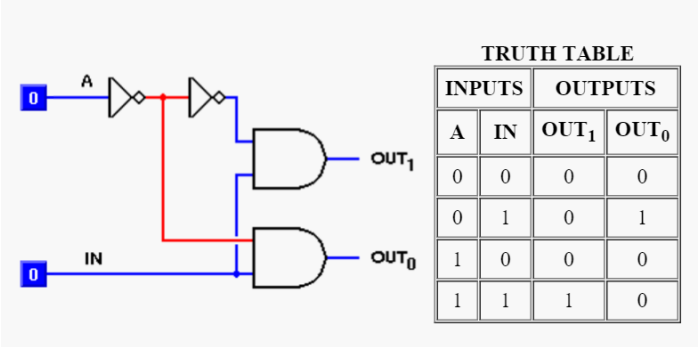
Below is the initial structure of the counter, where you can see the four outputs from the four JK flip flops, the clock, the four JK flip flop ICs and the Arduino power supply. A video of the counter is available as well. A major issue this the final outcome of the counter is that fact that I could never get the outputs to be in sync with the automatic clock. You can see the ripple effect somewhat take place, but with a lack of stability and out of sync with the timer and clock pulses.




Ripple_Counter.mp4

1-2 Decoder/Demultiplexer

A decoder changes a code into a set of signals. A decoder takes an n-digit binary number and ultimately decodes it into 2^n data lines. I chose to build this 1-2 decoder, whose circuit diagram can be seen in figure 21, because it also act as a demultiplexer. A demultiplexer takes a single data input and selects which of the multiple outputs will receive the input signal. A demultiplexer does this by using an enabler, represented at the input line called IN in figure 21. To build the decoder/demultiplexer I only needed two NOT gates and two AND gates. I used a hex converter IC for the NOT gates and built the two AND gates using basic components. From the truth table of the decoder, which can also be seen in figure 21, whenever the enable is zero, the decoder has no true output. Whenever, the enabler has a true value however, the decoder is allowed to produce a true output. This true output is dependent on the value of the input of the decoder. If the input is high, the output of the decoder is true for output one. This scenario can be seen in figure 22, where the light up LED represents the high value of output one. When the enable is high and the input is low, the decoder outputs a high value for output zero, this result is shown in figure 23.



TRUTH TABLE

INPUTS		OUTPUTS	
A	IN	OUT_1	OUT_0
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	0

Figure 21: Truth table and circuit diagram for decoder/demultiplexer

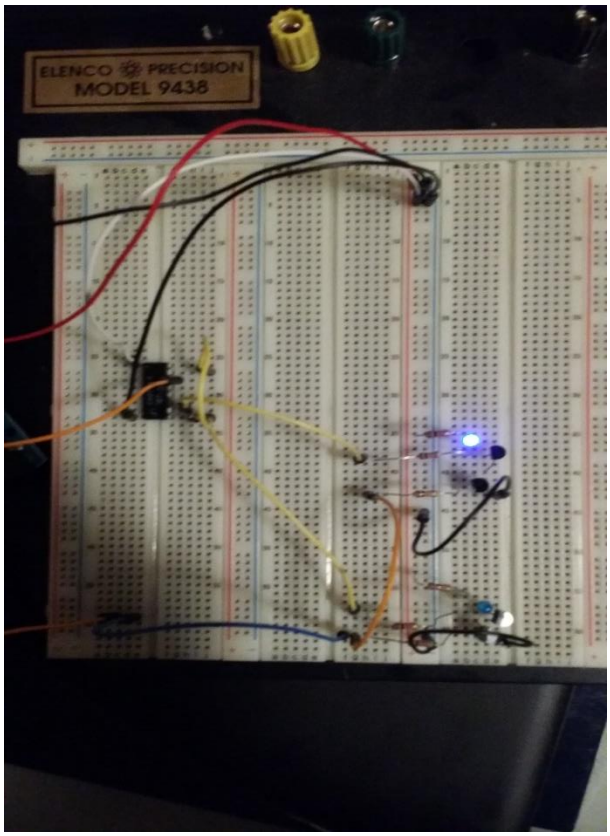


Figure 22: Implementation of decoder with high input

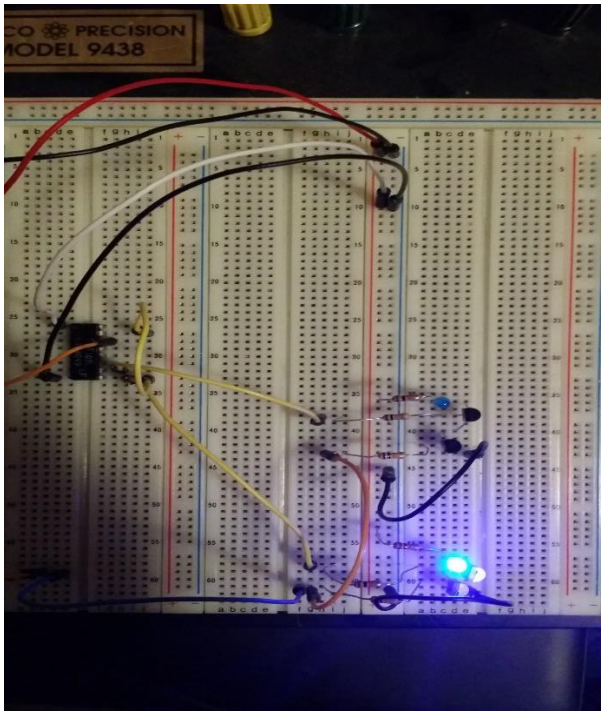
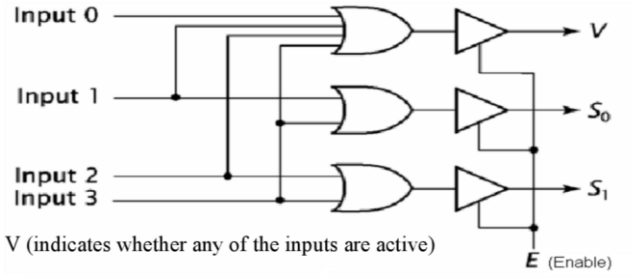


Figure 23: Implementation of decoder with low input

Encoder

An encoder, obviously, has the opposite function as a decoder. Where and encoder takes 2^n input lines and outputs n lines. I modelled my implementation of an encoder after the circuit diagram shown in figure 24. This encoder requires nothing by OR gates, five OR gates to be exact. To do this, I used the quad OR IC and built the additional OR gate using two transistors. The circuit diagram for the additional OR gate can be seen in figure 25. From the figure, you can see how if either one or both of the transistors were saturated with power, the power supply would be able to access the output. If neither of the transistors had power from the inputs however, the power would never reach the output or the ground and result in a false output from the OR gate. Also, assume for this encoder that the enable is always high because I did not build an enable into this circuit.



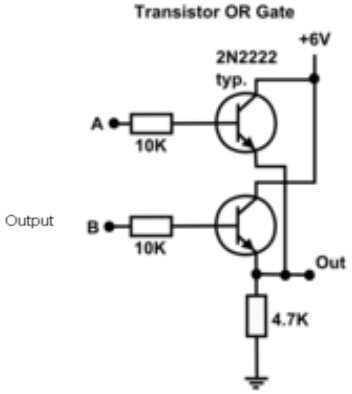
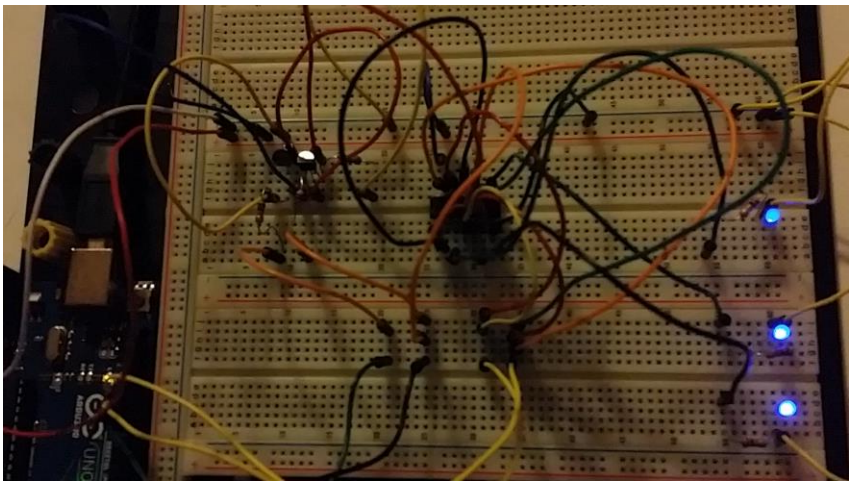


Figure 24: Circuit diagram for encoder

Figure 25: Circuit diagram for OR gate

Below you can view the initial structure of the encoder circuit and access a video of the varying outputs, the inputs of which were systemically changed with the Arduino pins. You can follow the outputs of the encoder with referring to the truth table of an encoder in figure 26.



I_0	I_1	I_2	I_3	E	S_1	S_2	V
X	X	X	X	0	Z	Z	Z
0	0	0	0	1	0	0	0
1	0	0	0	1	0	0	1
0	1	0	0	1	0	1	1
0	0	1	0	1	1	0	1
0	0	0	1	1	1	1	1

Figure 26: Truth table for encoder

Multiplexer

A multiplexer is a device that selects one of several inputs to be the output. The selection of this input to be the single output depends on the value of the selectors. I chose to build a multiplexer that used two selectors to pick between four inputs to be the single output. Figure 27 shows the circuit diagram for the multiplexer. To construct this multiplexer I needed two NOT gates, eight AND gates and two OR gates. I used the hex converter IC for the NOT gates, the quad OR IC for the OR gates, and the quad AND IC and four AND gates made from two transistors each for the AND gates. The truth table for the multiplexer, shown in figure 28, shows that when the selectors are both low, for instance, the multiplexer with output the first input. If both selectors are high, for instance, the multiplexer will output the fourth input.

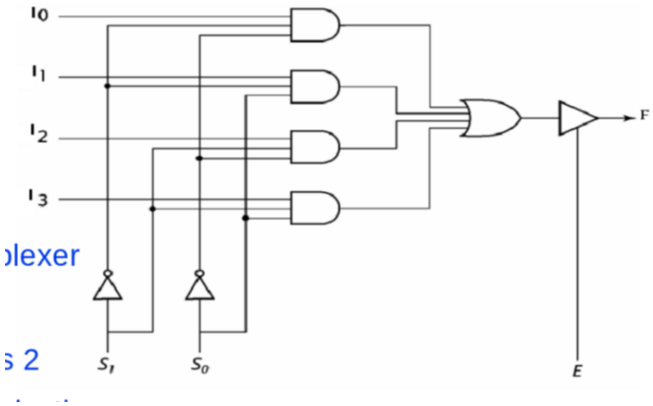


Figure 26: Circuit diagram for multiplexer

S_1	S_0	F
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Figure 26: Truth table for multiplexer

Unfortunately, I do not have any pictures or video to show for the multiplexer. This is because I brought the multiplexer circuit board in to demonstrate physically during the presentation for this project and dissembled it right after. Forgetting to make any video of its functionality.