DESIGN PROJECT

PROBLEM:

The objective of this design project is to design an FSM that will cycle through my student number via the use of flip-flops and logic gates.

ANALYTICAL IMPLEMENTATION:

To begin, I will create a **state transition table** that shows my BCD Student Number (400312849) in terms of the bits **Q1, Q2, Q3, and Q4**, as well as a counter **C1** that will distinguish between repeated digits, as well as the required states for each flip-flop's inputs.

#	Q1	Q2	Q3	Q4	C1	J1	!K1	J2	!K2	J3	!K3	J4	!K4	JC1	!KC1
4	0	1	0	0	x	0	x	x	0	0	x	0	x	0	0
0	0	0	0	0	0	0	x	0	x	0	x	0	x	1	x
0	0	0	0	0	1	0	x	0	x	1	x	1	x	x	x
3	0	0	1	1	x	0	x	0	x	x	0	x	1	x	x
1	0	0	0	1	x	0	x	0	x	1	x	x	0	x	x
2	0	0	1	0	x	1	x	0	x	x	0	0	x	x	x
8	1	0	0	0	x	x	0	1	x	0	x	0	x	x	x
4	0	1	0	0	x	1	x	x	0	0	x	1	x	x	x
9	1	0	0	1	x	x	0	0	x	0	x	x	0	x	x

While this is the most **robust** implementation of this design, I must make an effort to simplify the logic so that this FSM will be easier to implement physically. Unfortunately, a few options to simplify this further are **not** available to me. The options that are **not** available include having bits whose state are equal to each other (in my case Q1 ≠ Q2 ≠ Q3 ≠ Q4), and I have no bits that are all one value, so I can't replace any given bit with just a hook up to VCC or GND, and from what I can tell, no bits are uniquely determined by any combination of the other bits.

However, there are still simplifications that I can make. In order to simplify this further, I will change the 'don't care' states of C1 to **unique** states, and I will try to make it so that as best as I can (since there are only 8 numbers, one state will not be able to alternate from the last).

#	Q1	Q2	Q3	Q4	C1	J1	!K1	J2	!K2	J3	!K3	J4	!K4	JC1	!KC1
4	0	1	0	0	0	0	x	x	0	0	x	0	x	1	x
0	0	0	0	0	1	0	x	0	x	0	x	0	x	x	0
0	0	0	0	0	0	0	x	0	x	1	x	1	x	1	x
3	0	0	1	1	1	0	x	0	x	x	0	x	1	x	0
1	0	0	0	1	0	0	x	0	x	1	x	x	0	1	x
2	0	0	1	0	1	1	x	0	x	x	0	0	x	x	0
8	1	0	0	0	0	x	0	1	x	0	x	0	x	1	x
4	0	1	0	0	1	1	x	x	0	0	x	1	x	x	0
9	1	0	0	1	0	x	0	0	x	0	x	x	0	0	x

From this state transition table, I can determine the necessary inputs (J and !K) for each of my JK flip-flops.

#	Q1	Q2	Q3	Q4	C1	J1	!K1	J2	!K2	J3	!K3	J4	!K4	JC1	!KC1
4	0	1	0	0	0	0	x	x	0	0	x	0	x	1	x
0	0	0	0	0	1	0	x	0	x	0	x	0	x	x	0
0	0	0	0	0	0	0	x	0	x	1	x	1	x	1	x
3	0	0	1	1	1	0	x	0	x	x	0	x	1	x	0
1	0	0	0	1	0	0	x	0	x	1	x	x	0	1	x
2	0	0	1	0	1	1	x	0	x	x	0	0	x	x	0
8	1	0	0	0	0	x	0	1	x	0	x	0	x	1	x
4	0	1	0	0	1	1	x	x	0	0	x	1	x	x	0
9	1	0	0	1	0	x	0	1	x	0	x	x	0	0	x

For my K-Maps, I decided to use a mirror layout.

FLIP FLOP 1:

!K1 = 0

SoP implementation of J1:
J1 = (Q3*!Q4) + (Q2*C1)

Q1Q2/Q3Q4C1	000	001	011	010	110	111	101	100
00	0	0	x	0	x	0	1	x
01	0	1	x	x	x	x	x	x
11	x	x	x	x	x	x	x	x
10	x	x	x	x	x	x	x	x

PoS implementation of J1:
J1 = (Q2 + Q3)*(!Q4)*(C1)

Q1Q2/Q3Q4C1	000	001	011	010	110	111	101	100
00	0	0	x	0	x	0	1	x
01	0	1	x	x	x	x	x	x
11	x	x	x	x	x	x	x	x
10	x	x	x	x	x	x	x	x

For flip flop 1, the **SoP** implementation requires **2 AND gates, and an OR gate.** The **PoS** implementation also requires **2 AND gates, and an OR gate.** Therefore, neither implementation is inherently any better or easier than the other, however, if later on I find an input for a different flip flop that uses one of the same operations, I will choose the implementation that matches that. Luckily, !K1 is always a 0 or "don't care", so it can just be hooked up directly to GND.

FLIP FLOP 2:

!K2 = 0

J2 = Q1

For Flip Flop 2, !K2 is always a 0 or "don't care", so it can just be hooked up directly to GND, and J2 is always equal to Q1.

FLIP FLOP 3:

!K3 = 0

SoP implementation of J3:
J3 = !Q1*!Q2*!C1

Q1Q2/Q3Q4C1	**000**	**001**	**011**	**010**	**110**	**111**	**101**	**100**
00	1	0	x	1	x	x	x	x
01	0	0	x	x	x	x	x	x
11	x	x	x	x	x	x	x	x
10	0	x	x	0	x	x	x	x

PoS implementation of J3:
J3 = !Q1*!Q2*!C1

Q1Q2/Q3Q4C1	**000**	**001**	**011**	**010**	**110**	**111**	**101**	**100**
00	1	0	x	1	x	x	x	x
01	0	0	x	x	x	x	x	x
11	x	x	x	x	x	x	x	x
10	0	x	x	0	x	x	x	x

For flip flop 3, the **SoP** implementation requires **2 AND gates.** The **PoS** implementation also requires **2 AND gates.** Therefore, both implementations are the same, so it is trivial which one I choose. Again, !K3 is always a 0 or "don't care", so it can just be hooked up directly to GND.

FLIP FLOP 4:

!K4 = Q3

SoP implementation of J4:
J4 = (!Q1*!Q2*!C1) + (Q2*C1)

Q1Q2/Q3Q4C1	000	001	011	010	110	111	101	100
00	1	0	x	x	x	x	0	x
01	0	1	x	x	x	x	x	x
11	x	x	x	x	x	x	x	x
10	0	x	x	x	x	x	x	x

PoS implementation of J4:
J4 = (Q2 + !C1)*(!Q2 + C1)*(!Q1)

Q1Q2/Q3Q4C1	000	001	011	010	110	111	101	100
00	1	0	x	x	x	x	0	x
01	0	1	x	x	x	x	x	x
11	x	x	x	x	x	x	x	x
10	0	x	x	x	x	x	x	x

For flip flop 4, the **J4 SoP** implementation requires **3 AND gates and an OR gate.** The **J4 PoS** implementation also requires **2 AND gates and 2 OR gates.** I will determine which is more optimal in terms of the rest of the design at the end of this section. In this case, **!K4 = Q3.**

FLIP FLOP 5:

!KC1 = 0

SoP implementation of JC1:
JC1 = !Q1+ !Q4

Q1Q2/Q3Q4C1	000	001	011	010	110	111	101	100
00	1	x	x	1	x	x	x	x
01	1	x	x	x	x	x	x	x
11	x	x	x	x	x	x	x	x
10	1	x	x	0	x	x	x	x

PoS implementation of JC1:
JC1 = !Q1 + !Q4

Q1Q2/Q3Q4C1	**000**	**001**	**011**	**010**	**110**	**111**	**101**	**100**
00	1	x	x	1	x	x	x	x
01	1	x	x	x	x	x	x	x
11	x	x	x	x	x	x	x	x
10	1	x	x	0	x	x	x	x

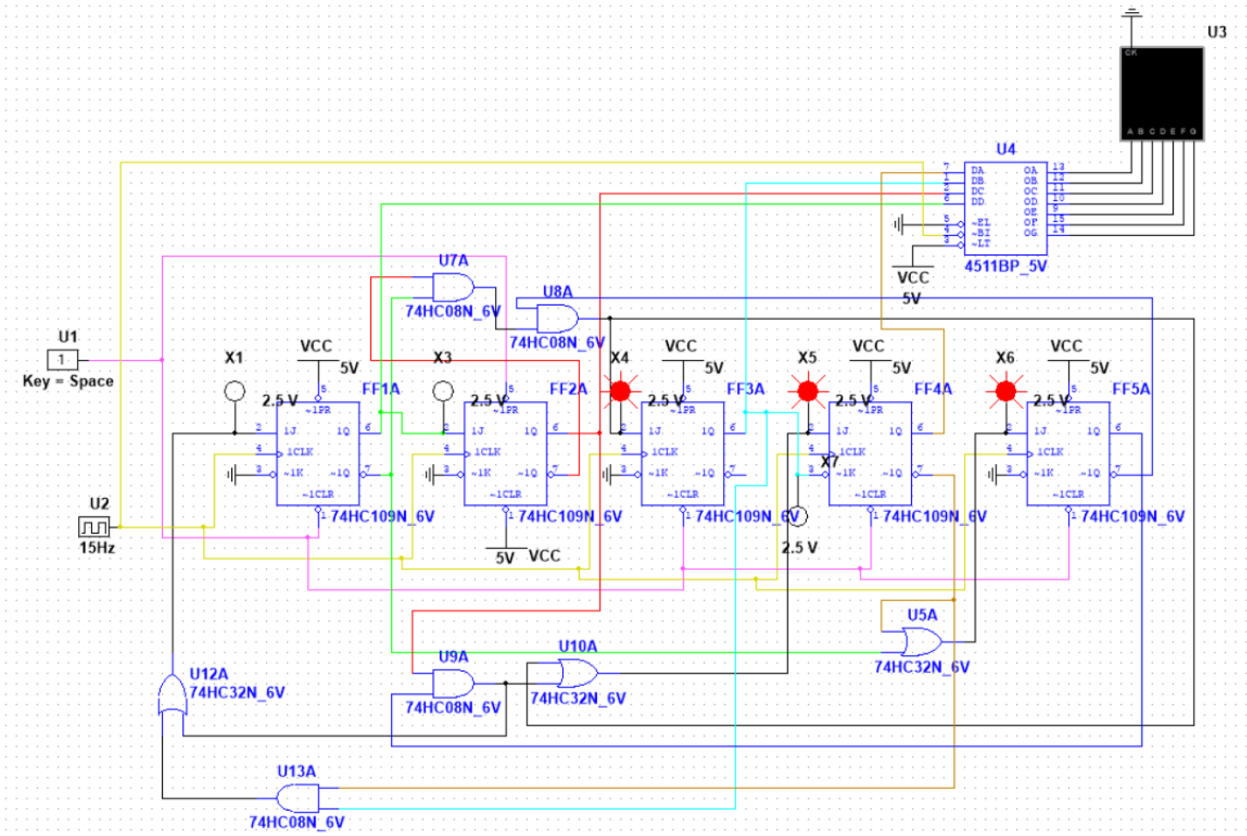
For flip flop 5, the **SoP** implementation requires **1 OR gate.** The **PoS** implementation also requires **1 OR gate.** Therefore, both implementations are the same, so it is trivial which one I choose. Again, !KC1 is always a 0 or "don't care", so it can just be hooked up directly to GND.

ANALYTICAL SUMMARY:

Below is a table outlining the most optimal implementation of this FSM based on the SoP and Pos implementations. In total, I will need: **4 AND gates and 3 OR gates.**

Input	**SoP Implementation**	**PoS Implementation**	**Optimal Implementation**
J1	(Q3*!Q4) + (Q2*C1)	(Q2 + Q3)*(!Q4)*(C1)	(Q3*!Q4) + (Q2*C1)
!K1	0	0	0
J2	Q1	Q1	Q1
!K2	0	0	0
J3	!Q1*!Q2*!C1	!Q1*!Q2*!C1	!Q1*!Q2*!C1
!K3	0	0	0
J4	(!Q1*!Q2*!C1) + (Q2*C1)	(Q2 + !C1)*(!Q2 + C1)*(!Q1)	(Q1*!Q2*!C1) + (Q2*C1)
!K4	Q3	Q3	Q3
JC1	!Q1+ !Q4	!Q1+ !Q4	!Q1+ !Q4
!KC1	0	0	0

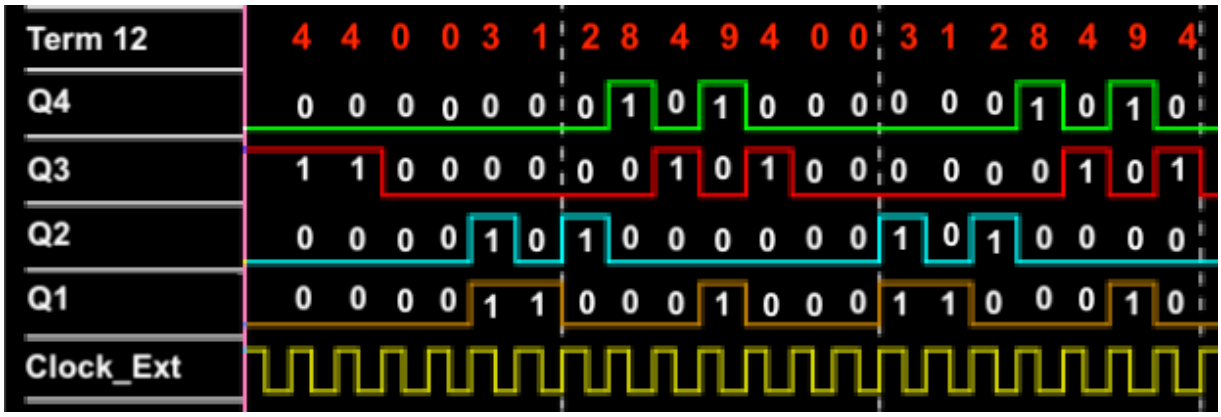
MULTISIM IMPLEMENTATION:



COLOUR CODE:

Black: Either a regular meaningless wire connection (e.g. to 7SD), **or** the result of a logic operation
Pink: Pre-set/Clear/Force initial state hookups
Yellow: Clock signal
Green: FF1 outputs (Q1, !Q1)
Red: FF2 outputs (Q2, !Q2)
Cyan: FF3 outputs (Q3, !Q3)
Brown: FF4 outputs (Q4, !Q4)
Blue: FF5 outputs (C1, !C1)

OUTPUT TIMING DIAGRAM:



In the above timing diagram, my student number is numbered in red, while each of the respective bits is labelled. Please note that the double 4 at the beginning is **NOT** an error, it is the 4-state repeating itself before I un-force the value (e.g. since I have to force the beginning state, it cycles on that value until I un-force it by switching the respective PR/CLR pins from a 0 back to a 1.)

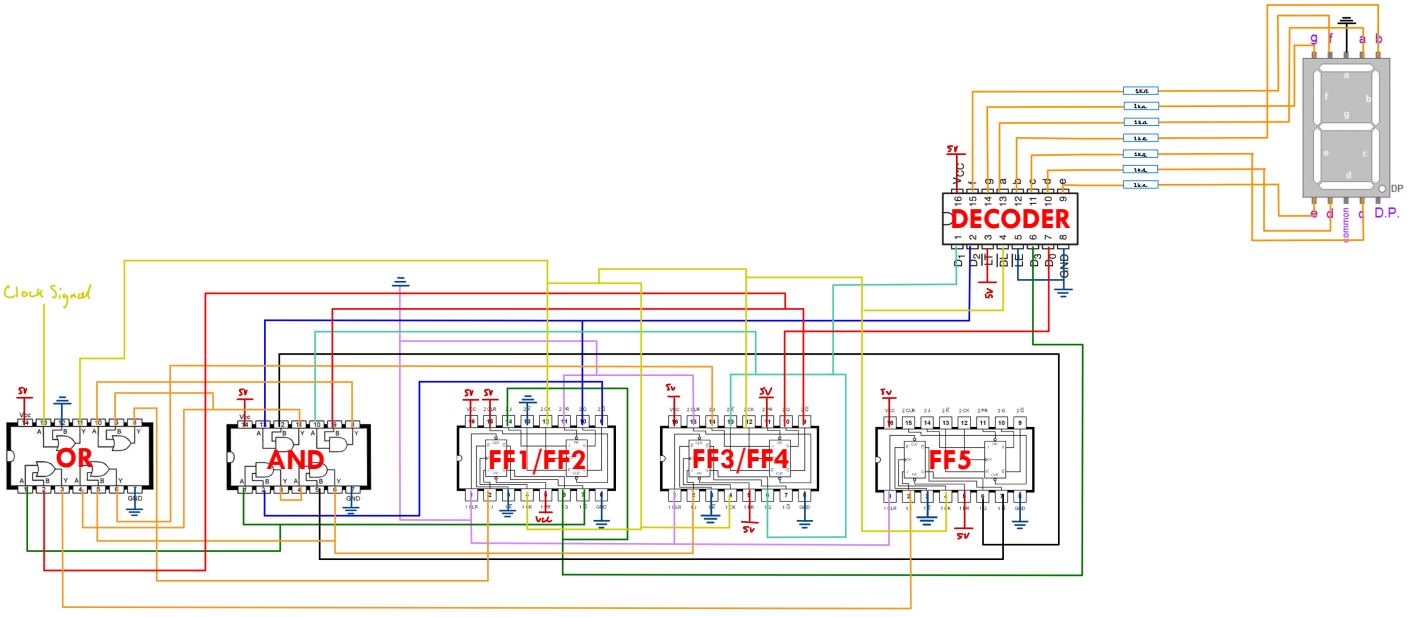
LINK TO YOUTUBE VIDEO OF MULTISIM RUNNING:

https://youtu.be/Bklr8uTOh8Y?t=79

PHYSICAL IMPLEMENTATION:

PRE-BUILD DIAGRAM:

PLEASE NOTE: the teal coloured wire on the drawn diagram represents the white wire on my real build (since white doesn't show up on a white background)

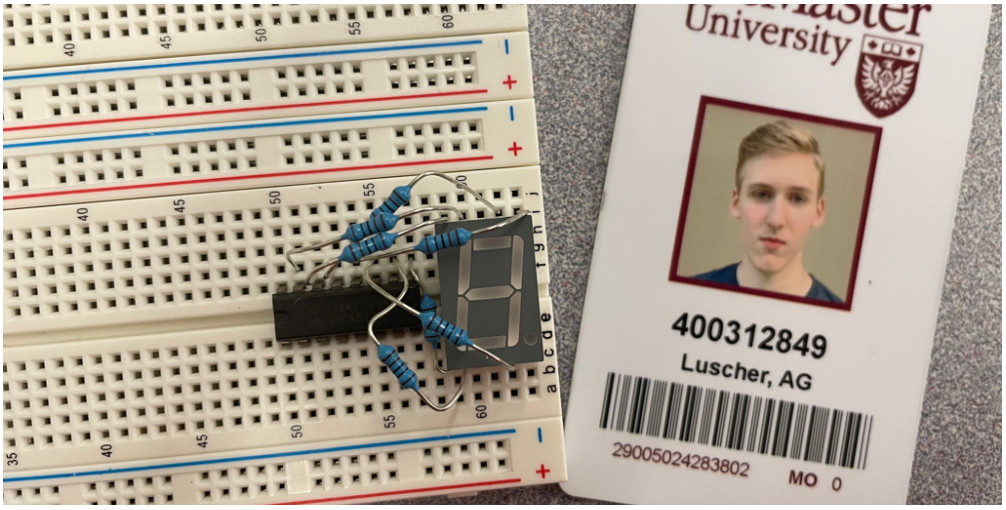


COLOUR CODE:

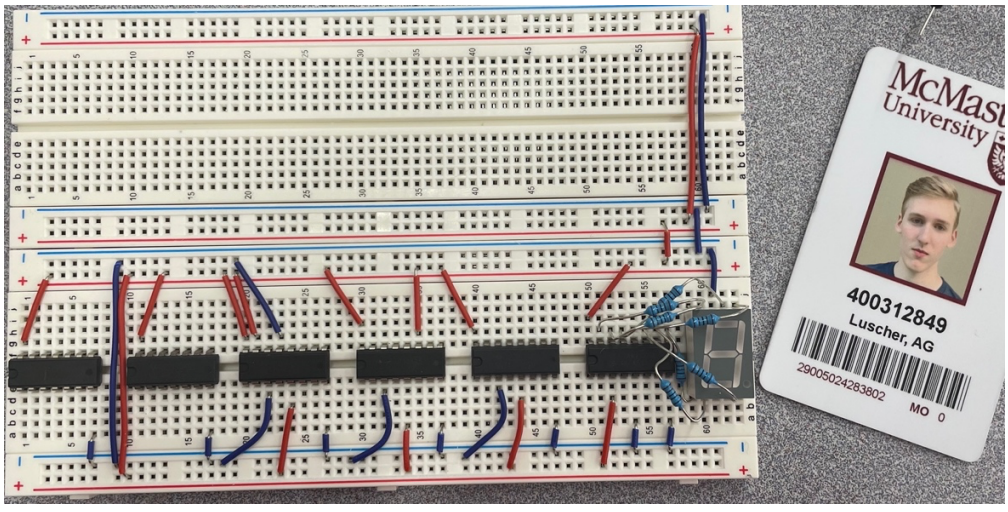
Yellow: clock signal
Purple: forced beginning state connections
Orange: result of **any logic operation** *OR* a meaningless connection (e.g. from the decoder to the 7SD)
Green: FF1 output (Q1, !Q1)
Royal Blue: FF2 output (Q2, !Q2)
White (Teal on drawn diagram): FF3 output (Q3)
Bright Red: FF4 output (Q4, !Q4)
Black: FF5 output (C1, !C1)
Dark Red: 5V VCC
Dark Blue: GND

BUILD METHOD:

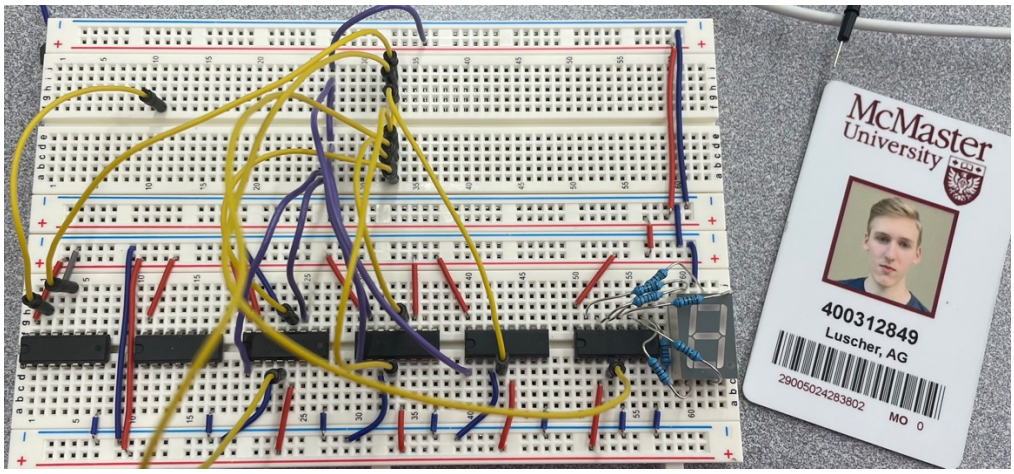
First, I connected the 7SD to the decoder chip, and connected the respective pins using **1kΩ resistors.**



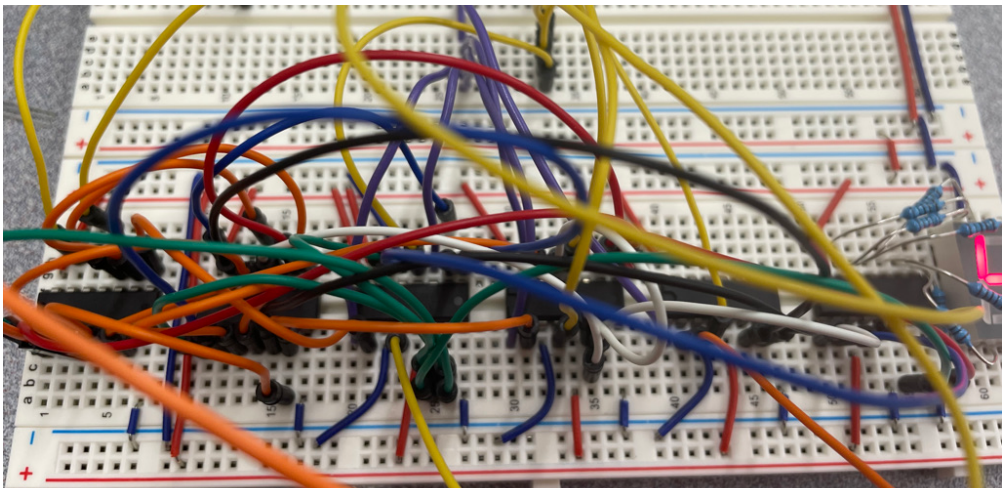
Next, I placed all the **OR, AND, and JK Flip-Flop** chips on the board, and wired up their respective VCC, GND, and PR/CLR pins that won't be changing, as well as any steady-state inputs to any of the chips.



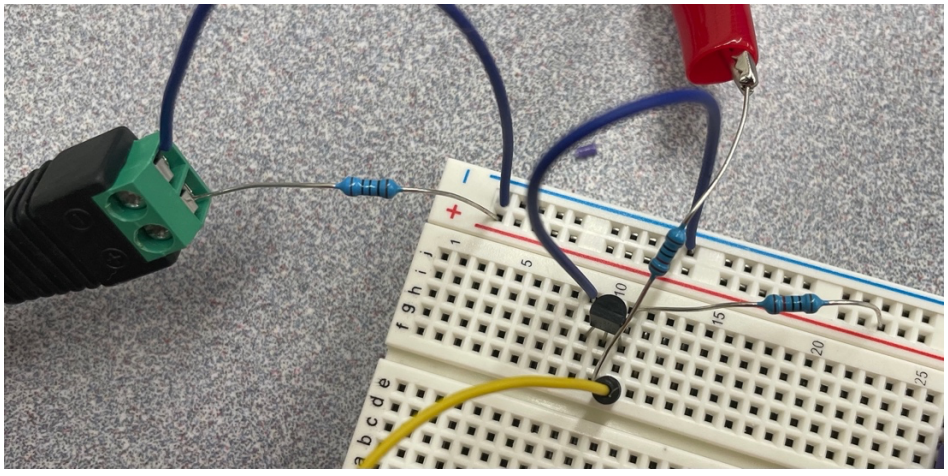
Next, I wired up all the clock signals as well as the PR/CLR pins that will require forcing.



Then, I wired up all the logic and flipflop connections.

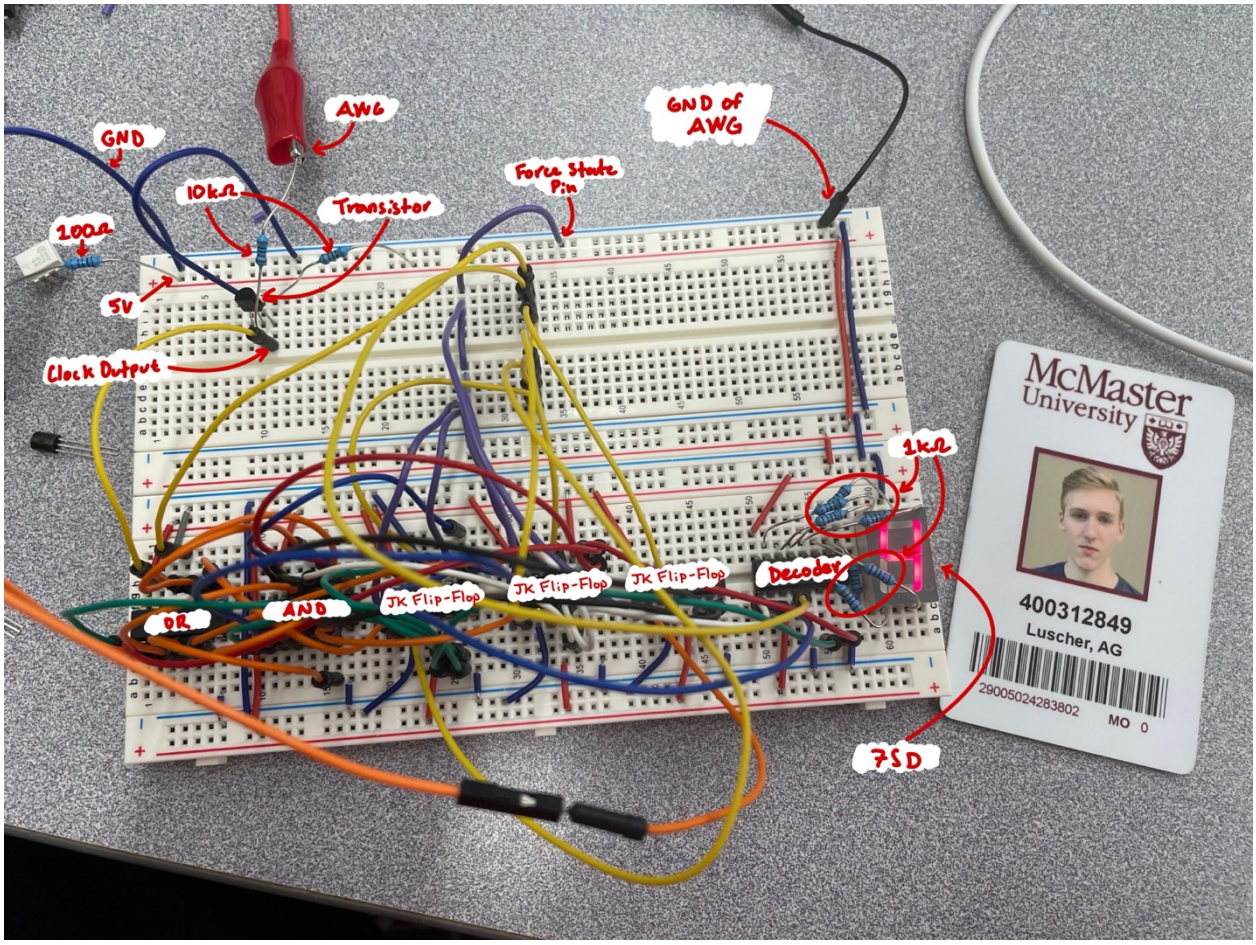


I wired up the clock signal like so, where the yellow wire ran to an OR gate with the other input tied to GND so that I could have a square input wave. The red alligator clip was the output from the HANTEK.



FINAL RESULT:

The aforementioned steps left me with a final build that looked like this:



The colour code here was the same as the drawn one.

Fortunately, **my circuit worked on the first try**, and I was saved from any *real* debugging. Initially, I had tried to test that the logic was working by manually moving a DC input from high to low for the clock signal, but soon realized that the logic chips expect steady intervals between the clock signal, which a I cannot easily provide by hand.

PRESENTATION LINK (INCLUDES VIDEO OF PHYSICAL RUNNING):

https://youtu.be/BkIr8uTOh8Y