ENGPHYS 2E04
Design Project
Sequential Logic Design

Dorian Knight, knighd7@mcmaster.ca, 400304437

December 10, 2021

Contents

Introduction

For the final project of ENGPHYS 2E04 us students have been asked to utilize our skills acquired in the digital section of the course to implement a sequential logic design that displays our student number on a 7 segment display. Since my student number is **400304437** I will work to implement a series of JK flip flops and logic gates such that by the end of this lab I will have analytically determined the layout, prototyped it in multisim and then physically implement it on my breadboard.

Goals for this lab

1. Analytical
    a. Generate truth table for current and future state, explain the don't care cases
    b. Generate and explain the K-Maps (use colour-coded boxes)
    c. Explain design choices and optimizations
    d. Compare with 2 other alternative designs/implementations in terms of number of gates and chips used
2. Multisim
    a. Validate circuit design using multisim via screen captured video showing simulation in progress and include an image and explanation of my implementation
    b. Wires need to be colour coded and explained
    c. Timing diagram should be included as well
3. Final Build and Presentation
    a. Assemble circuit and document the assembly process along the way
    b. Video of circuit functioning
    c. Wires are colour coded and colour code is explained

Analytical

Understanding the nature of the problem

Before jumping right in to developing the sequential logic I must understand how each state transitions into the next state – if there are any patterns that I notice in my student number I may be able to simplify my sequential logic and make the lab easier right from the start.

The table below has Q1-Q4 representing the student number in binary where Q4 is the MSB and the Q1 is the LSB. Memory 1 and Memory 2 will keep track of which instance of a "4" is currently happening. Eg, first 4, second 4 or third 4.

It is important to keep track of how many instances of the same number has occurred because while the number may be the same, the next state may be different. To implement a sequential logic we must find a way to distinguish between all instances of the same number.

Individual Number	Flip flop binary number outputs				Memory outputs	
	Q4	Q3	Q2	Q1	Memory 1	Memory 2
4	0	1	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	1
3	0	0	1	1	X	0
0	0	0	0	0	1	X
4	0	1	0	0	0	1
4	0	1	0	0	1	X
3	0	0	1	1	X	1
7	0	1	1	1	X	X

Table 1 Student number with memory states – X indicates don't care case

Simplifying the problem

Upon first glance there are a few simplifications that can be made to the system.

Firstly, since I only have numbers below 8, my Q4 is always zero. This means that for the Q4 input into the BCD to 7SD decoder chip can be hooked up to ground.

I also notice that my Q2 and Q1 only change together. This means that I can use the same JK flip flop output to fulfill the purpose of both Q1 and Q2.

It is also worth noting that my student number only has 4 numbers in it where 3 of them repeat either 2 or 3 times. This means that compared to other students I may have less don't care cases in my memory outputs because most of my digits repeat and I'll be relying more on memory outputs to help dictate my sequential logic.

Memory Optimization

I'm not 100% certain but I would imagine that other students have more don't care statements in their memory outputs. Since my student number only contains four distinct numbers (4,0,3,7) and every number appears more than once (except 7), my memory flip flops have an important role to play in regulating the sequential logic. The only don't care conditions I have are as follows. In Memory 1 when

outputting the number three, since it only appears twice and Memory 2 is able to handle that responsibility by itself. In Memory 1 and 2 when outputting the number 7 (since it doesn't repeat) and on the third occurrence of 4 and 0 where Memory 2 could either be a 0 or 1 since having Memory 1 being a 1 distinguishes this occurrence from the first and second repetitions as Memory 1 would have previously been 0.

Unfortunately, I can't make any more don't care cases without sacrificing the integrity of the memory system. This is why I only have 6 don't care cases. This may make my logic in the K-mapping section more intense than other projects just because in general, less don't care cases makes the logic more complicated.

Updated Table

Individual Number	Q2 (XXXX)	Q1 (XXXX)	Q3 (Memory 1)	Q4 (Memory 2)
4	1	0	0	0
0	0	0	0	0
0	0	0	0	1
3	0	1	X	0
0	0	0	1	X
4	1	0	0	1
4	1	0	1	X
3	0	1	X	1
7	1	1	X	X

Table 2 Revised table utilizing the aforementioned optimizations

The names have changed from the first table in the following ways:

- Q4 is now constantly connected to ground so therefore Memory 2 now has the name Q4
- Q1 and Q2 are represented by the same output therefore the combined output has now been renamed Q1
- Since the name Q2 is no longer being used Q3 how now become Q2
- Because the name Q3 is no longer being used, memory 1 has taken on the name Q3

Now that I've simplified my problem down from needing 6 flip flops down to only needing 4, I can start by producing K-maps to further understand and design my logical sequence.

Designing The Logic Sequence – Considering Flip Flop States

Before diving any deeper I must understand what inputs are needed to invoke certain actions from a flip flop. Up until this point we have worked with JK flip flop truth table which is good for logic sequence analysis however now that I'm designing my own logic sequence I need to understand what inputs are required to get my next desired output.

JK Truth table

J	K	!K	Qnext
0	0	1	Q_n (hold)
0	1	0	0 (reset)
1	0	1	1 (set)
1	1	0	!Q_n(toggle)

Table 3 JK flip flop truth table

JK Excitation table

Q_n	Q_{n+1}	J	K	!K	Invoked action
0	0	0	X	X	Reset or Hold
0	1	1	X	X	Set or Toggle
1	0	X	1	0	Reset or Toggle
1	1	X	0	1	Set or Hold

Table 4 JK flip flop excitation table

In the JK excitation table the Xs indicate don't care cases. When Q is 0 and we want the output to stay at 0 we can either force the switch to hold or reset. Holding would keep the output at zero which is the desired output but a reset would also do the same thing therefore it doesn't matter what the value of K is, both 0 and 1 will have the same effect.

Truth Table Q1

Current state	Next state	Required flip flop action	Required flip flop input	
Q1			J1	K1
0	0	Reset or Hold	0	X
0	0	Reset or Hold	0	X
0	1	Set or Toggle	1	X
1	0	Reset or Toggle	X	1
0	0	Reset or Hold	0	X
0	0	Reset or Hold	0	X
0	1	Set or Toggle	1	X
1	1	Set or Hold	X	0
1	0	Reset or Toggle	X	1

Table 5 Truth table for desired logic for Q1

Truth Table Q2

Current state	Next state	Required flip flop action	Required flip flop input	
Q2			J2	K2
1	0	Reset or Toggle	X	1
0	0	Reset or Hold	0	X
0	0	Reset or Hold	0	X
0	0	Reset or Hold	0	X
0	1	Set or Toggle	1	X
1	1	Set or Hold	X	0
1	0	Reset or Toggle	X	1
0	1	Set or Toggle	1	X
1	1	Set or Hold	X	0

Table 6 Truth table for desired logic for Q2

Truth Table Q3

Current state	Next state	Required flip flop action	Required flip flop input	
Q3			J3	K3
0	0	Reset or Hold	0	X
0	0	Reset or Hold	0	X
0	X	N/A	X	X
X	1	Set	1	0
1	0	Reset or Toggle	X	1
0	1	Set or Toggle	1	X
1	X	N/A	X	X
X	X	N/A	X	X
X	0	Reset	0	1

Table 7 Truth table for desired logic for Q3 (memory 1) – NA=Not Applicable

Truth Table Q4

Current state	Next state	Required flip flop action	Required flip flop input	
Q4			J4	K4
0	0	Reset or Hold	0	X
0	1	Set or Toggle	1	X
1	0	Reset or Toggle	X	1
0	X	N/A	X	X
X	1	Set	1	0
1	X	N/A	X	X
X	1	Set	1	0
1	X	N/A	X	X
X	0	Reset	0	1

Table 8 Truth table for desired logic for Q4 (memory 2)

Complete Truth Table For All Flip Flops

Q1		Q2		Q3		Q4		Required flip flop input							
Current state	Next state	Current state	Next state	Current state	Next state	Current state	Next state	J1	K1	J2	K2	J3	K3	J4	K4
0	0	1	0	0	0	0	0	0	X	X	1	0	X	0	X
0	0	0	0	0	0	0	1	0	X	0	X	0	X	1	X
0	1	0	0	0	X	1	0	1	X	0	X	X	X	X	1
1	0	0	0	X	1	0	X	X	1	0	X	1	0	X	X
0	0	0	1	1	0	X	1	0	X	1	X	X	1	1	0
0	0	1	1	0	1	1	X	0	X	X	0	1	X	X	X
0	1	1	0	1	X	X	1	1	X	X	1	X	X	1	0
1	1	0	1	X	X	1	X	X	0	1	X	X	X	X	X
1	0	1	1	X	0	X	0	X	1	X	0	0	1	0	1

Table 9 Combined truth table for all flip flop outputs and the required inputs to change the current state into the desired next state

Since the JK flip flop takes inputs for !K rather than K, I'll invert all of my K inputs such that I can get the corresponding logic that will work for the circuit components that we were given. Don't care cases don't need to be changed because they don't matter.

Q1		Q2		Q3		Q4		Required flip flop input							
Current state	Next state	Current state	Next state	Current state	Next state	Current state	Next state	J1	!K1	J2	!K2	J3	!K3	J4	!K4
0	0	1	0	0	0	0	0	0	X	X	0	0	X	0	X
0	0	0	0	0	0	0	1	0	X	0	X	0	X	1	X
0	1	0	0	0	X	1	0	1	X	0	X	X	X	X	0
1	0	0	0	X	1	0	X	X	0	0	X	1	1	X	X
0	0	0	1	1	0	X	1	0	X	1	X	X	0	1	1
0	0	1	1	0	1	1	X	0	X	X	1	1	X	X	X
0	1	1	0	1	X	X	1	1	X	X	0	X	X	1	1
1	1	0	1	X	X	1	X	X	1	1	X	X	X	X	X
1	0	1	1	X	0	X	0	X	0	X	1	0	0	0	0

Table 10 Because the JK flip flop takes !K instead of K, the table above was created for k-mapping

Now that we have the truth table for all of our J and K inputs for each flip flop, we can start making K-maps to determine the logic that is required to produce the input values that we want.

K-mapping

<table>
<tr><th></th><th>Subgroup of K mapping</th><th>J</th><th>K</th></tr>
<tr><td rowspan="2">JK1</td><td>SOP</td><td>J1 K-map
<table><tr><th>Q3Q4/Q1Q2</th><th>00</th><th>01</th><th>11</th><th>10</th></tr><tr><td>00</td><td>0</td><td>0</td><td>X</td><td>X</td></tr><tr><td>01</td><td>1</td><td>0</td><td>X</td><td>X</td></tr><tr><td>11</td><td>0</td><td>1</td><td>X</td><td>X</td></tr><tr><td>10</td><td>0</td><td>1</td><td>X</td><td>X</td></tr></table>Table 11 SOP K-map for J1

AND OR NOT Implementation
$(\overline{Q2}\,\overline{Q3}Q4) + (Q2Q3)$
Requires: 2 NOT, 3 AND, 1 OR
NAND Implementation
$\overline{(\overline{Q2}\,\overline{Q3}\,Q4)(\overline{Q2Q3})}$
Requires: 9 NAND</td><td>!K1 K-map
<table><tr><th>Q3Q4/Q1Q2</th><th>00</th><th>01</th><th>11</th><th>10</th></tr><tr><td>00</td><td>X</td><td>X</td><td>0</td><td>0</td></tr><tr><td>01</td><td>X</td><td>X</td><td>0</td><td>1</td></tr><tr><td>11</td><td>X</td><td>X</td><td>0</td><td>1</td></tr><tr><td>10</td><td>X</td><td>X</td><td>0</td><td>0</td></tr></table>Table 12 SOP K-map for !K1

AND OR NOT Implementation
$\overline{Q2}Q4$
Requires: 1 NOT, 1 AND
NAND Implementation
$\overline{Q2}Q4$(SAME)
Requires: 3 NAND</td></tr>
<tr><td>POS</td><td>J1 K-map
<table><tr><th>Q3Q4/Q1Q2</th><th>00</th><th>01</th><th>11</th><th>10</th></tr><tr><td>00</td><td>0</td><td>0</td><td>X</td><td>X</td></tr><tr><td>01</td><td>1</td><td>0</td><td>X</td><td>X</td></tr><tr><td>11</td><td>0</td><td>1</td><td>X</td><td>X</td></tr><tr><td>10</td><td>0</td><td>1</td><td>X</td><td>X</td></tr></table>Table 13 POS K-map for J1

AND OR NOT Implementation
$(Q2 + \overline{Q3})(Q3 + Q4)(\overline{Q2} + Q3)$
Requires: 2 NOT, 3 OR, 3 AND
NAND Implementation
$(\overline{Q2}\overline{Q3})(\overline{Q3}\,\overline{Q4})(\overline{Q2}\overline{Q3})$
Requires: 10 NAND</td><td>!K1 K-map
<table><tr><th>Q3Q4/Q1Q2</th><th>00</th><th>01</th><th>11</th><th>10</th></tr><tr><td>00</td><td>X</td><td>X</td><td>0</td><td>0</td></tr><tr><td>01</td><td>X</td><td>X</td><td>0</td><td>1</td></tr><tr><td>11</td><td>X</td><td>X</td><td>0</td><td>1</td></tr><tr><td>10</td><td>X</td><td>X</td><td>0</td><td>0</td></tr></table>Table 14 POS K-map for !K1

AND OR NOT Implementation
$\overline{Q2}Q4$
Requires: 1 NOT, 1 AND
NAND Implementation
$\overline{Q2}Q4$
Requires: 3 NAND</td></tr>
<tr><td rowspan="2">JK2</td><td>SOP</td><td>J2 K-map
<table><tr><th>Q3Q4/Q1Q2</th><th>00</th><th>01</th><th>11</th><th>10</th></tr><tr><td>00</td><td>0</td><td>X</td><td>X</td><td>0</td></tr><tr><td>01</td><td>0</td><td>X</td><td>X</td><td>1</td></tr><tr><td>11</td><td>1</td><td>X</td><td>X</td><td>1</td></tr><tr><td>10</td><td>1</td><td>X</td><td>X</td><td>0</td></tr></table>Table 15 SOP K-map for J2

AND OR NOT Implementation
$\overline{Q1}Q3 + Q1Q4$
Requires: 1 NOT, 1 OR, 2 AND
NAND Implementation</td><td>!K2 K-map
<table><tr><th>Q3Q4/Q1Q2</th><th>00</th><th>01</th><th>11</th><th>10</th></tr><tr><td>00</td><td>X</td><td>0</td><td>1</td><td>X</td></tr><tr><td>01</td><td>X</td><td>1</td><td>1</td><td>X</td></tr><tr><td>11</td><td>X</td><td>0</td><td>1</td><td>X</td></tr><tr><td>10</td><td>X</td><td>0</td><td>1</td><td>X</td></tr></table>Table 16 SOP K-map for !K2

AND OR NOT Implementation
$Q1 + \overline{Q3}Q4$
Requires: 2 NOT, 1 OR, 1 AND
NAND Implementation</td></tr>
</table>

		$\overline{(\overline{\overline{Q1}Q3})(\overline{Q1Q4})}$ Requires: 4 NAND	$\overline{(\overline{Q1})(\overline{\overline{Q3}Q4})}$ Requires: 4 NAND
JK3	POS	J2 K-map Q3Q4/Q1Q2 | 00 | 01 | 11 | 10 00 | 0 | X | X | 0 01 | 0 | X | X | 1 11 | 1 | X | X | 1 10 | 1 | X | X | 0 *Table 17 POS K-map for J2* **AND OR NOT Implementation** $(Q1 + Q3)(\overline{Q1} + Q4)$ Requires: 1 NOT, 2 OR, 1 AND **NAND Implementation** $\overline{(\overline{Q1}\,\overline{Q3})(\overline{Q1\overline{Q4}})}$ Requires: 7 NAND	!K2 K-map Q3Q4/Q1Q2 | 00 | 01 | 11 | 10 00 | X | 0 | 1 | X 01 | X | 1 | 1 | X 11 | X | 0 | 1 | X 10 | X | 0 | 1 | X *Table 18 POS K-map for !K2* **AND OR NOT Implementation** $(Q1 + \overline{Q3})(Q1 + Q4)$ $Q1 + (\overline{Q3}Q4)$ Requires: 2 NOT, 1 OR, 1 AND **NAND Implementation** $\overline{\overline{Q1}(\overline{\overline{Q3}Q4})}$ Requires: 4 NAND
	SOP	J3 K-map Q3Q4/Q1Q2 | 00 | 01 | 11 | 10 00 | 0 | 0 | 0 | 1 01 | X | 1 | 0 | X 11 | X | X | 0 | X 10 | X | X | 0 | 1 *Table 19 SOP K-map for J3* **AND OR NOT Implementation** $\overline{Q1}Q4 + Q1\overline{Q2}$ Requires: 2 NOT, 1 OR, 2 AND **NAND Implementation** $\overline{(\overline{\overline{Q1}Q4})(\overline{Q1\overline{Q2}})}$ Requires: 5 NAND	!K3 K-map Q3Q4/Q1Q2 | 00 | 01 | 11 | 10 00 | X | X | 0 | 1 01 | X | X | 0 | X 11 | 0 | X | 0 | X 10 | 0 | X | 0 | 1 *Table 20 SOP K-map for !K3* **AND OR NOT Implementation** $Q1\overline{Q2}$ Requires: 1 NOT, 1 AND **NAND Implementation** $Q1\overline{Q2}$ Requires: 3 NAND
	POS	J3 K-map Q3Q4/Q1Q2 | 00 | 01 | 11 | 10 00 | 0 | 0 | 0 | 1 01 | X | 1 | 0 | X 11 | X | X | 0 | X 10 | X | X | 0 | 1 *Table 21 K-map for J3* **AND OR NOT Implementation** $(Q1 + Q4)(\overline{Q1} + \overline{Q2})$ Requires: 2 NOT, 2 OR, 1 AND **NAND Implementation** $\overline{(\overline{Q1}\,\overline{Q4})(\overline{Q1Q2})}$ Requires: 6 NAND	!K3 K-map Q3Q4/Q1Q2 | 00 | 01 | 11 | 10 00 | X | X | 0 | 1 01 | X | X | 0 | X 11 | 0 | X | 0 | X 10 | 0 | X | 0 | 1 *Table 22 POS K-map for !K3* **AND OR NOT Implementation** $Q1\overline{Q2}$ Requires: 1 NOT, 1 AND **NAND Implementation** $Q1\overline{Q2}$ Requires: 3 NAND

<table>
<tr><td rowspan="2">JK4</td><td>SOP</td><td>J4 K-map
<table><tr><th>Q3Q4/Q1Q2</th><th>00</th><th>01</th><th>11</th><th>10</th></tr><tr><td>00</td><td>1</td><td>0</td><td>0</td><td>X</td></tr><tr><td>01</td><td>X</td><td>X</td><td>0</td><td>X</td></tr><tr><td>11</td><td>1</td><td>1</td><td>0</td><td>X</td></tr><tr><td>10</td><td>1</td><td>1</td><td>0</td><td>X</td></tr></table>Table 23 SOP K-map for J4
AND OR NOT Implementation
$\overline{Q2} + \overline{Q1}Q3$
Requires: 2 NOT, 1 OR, 1 AND
NAND Implementation
$\overline{Q2(\overline{\overline{Q1}Q3})}$
Requires: 3 NAND</td><td>!K4 K-map
<table><tr><th>Q3Q4/Q1Q2</th><th>00</th><th>01</th><th>11</th><th>10</th></tr><tr><td>00</td><td>X</td><td>X</td><td>0</td><td>X</td></tr><tr><td>01</td><td>0</td><td>X</td><td>0</td><td>X</td></tr><tr><td>11</td><td>1</td><td>1</td><td>0</td><td>X</td></tr><tr><td>10</td><td>1</td><td>1</td><td>0</td><td>X</td></tr></table>Table 24 SOP K-map for !K4
AND OR NOT Implementation
$\overline{Q1}Q3$
Requires: 1 NOT, 1 AND
NAND Implementation
$\overline{Q1}Q3$
Requires: 1 NAND</td></tr>
<tr><td>POS</td><td>J4 K-map
<table><tr><th>Q3Q4/Q1Q2</th><th>00</th><th>01</th><th>11</th><th>10</th></tr><tr><td>00</td><td>1</td><td>0</td><td>0</td><td>X</td></tr><tr><td>01</td><td>X</td><td>X</td><td>0</td><td>X</td></tr><tr><td>11</td><td>1</td><td>1</td><td>0</td><td>X</td></tr><tr><td>10</td><td>1</td><td>1</td><td>0</td><td>X</td></tr></table>Table 25 POS K-map for J4
AND OR NOT Implementation
$(\overline{Q2} + Q3)(\overline{Q1})$
Requires: 2 NOT, 1 OR, 1 AND
NAND Implementation
$(\overline{Q2\overline{Q3}})(\overline{Q1})$
Requires: 5 NAND</td><td>!K4 K-map
<table><tr><th>Q3Q4/Q1Q2</th><th>00</th><th>01</th><th>11</th><th>10</th></tr><tr><td>00</td><td>X</td><td>X</td><td>0</td><td>X</td></tr><tr><td>01</td><td>0</td><td>X</td><td>0</td><td>X</td></tr><tr><td>11</td><td>1</td><td>1</td><td>0</td><td>X</td></tr><tr><td>10</td><td>1</td><td>1</td><td>0</td><td>X</td></tr></table>Table 26 POS K-map for !K4
AND OR NOT Implementation
$\overline{Q1}Q3$
Requires: 1 NOT, 1 AND
NAND Implementation
$\overline{Q1}Q3$
Requires: 2 NAND</td></tr>
</table>

Table 27 K-mapping for SOP and POS versions for each J and K input for each of the four flip flops showcasing both AND OR NOT and NAND implementations

Note: Chip count was made with the assumption that that !Qx doesn't exist. This is to demonstrate that I do know how to count NANDs needed in a case where I would have to create !Qx myself. Going forward in the next section and beyond, I use the actual number of NANDs needed as I won't need to invert the output myself, the chip does that for me thankfully.

Selecting which implementation to use

J1

For JK1 J input I've decided to use the **SOP AND OR NOT implementation** because it uses 2 fewer OR gates compared to the POS implementation and I decided to not use a NAND implementation because I plan to use NAND a lot for the other JK flip flops and I may run out of NAND gates if I'm not careful.

Requirements: 3 AND 1 OR

!K1

For JK1 !K input I've decided to use the **SOP and POS AND OR NOT implementation**. Since the flip flops output both QX and !QX, I don't need to use an additional NOT gate to invert the output. Using the AND OR NOT implementation allows me to consolidate the different kinds of chips I can use in order to make the wiring more organized. I'll be able to keep my wiring together as the 1 AND chip that I will use will be the same one as I used for the J input. The POS implementations are the same as SOP. I chose to not use the NAND implementation as it would be inefficient because I would be putting two NAND gates together just to function as an AND gate.

Requirements: 1 AND

J2

For JK2 J input I'm going to use the **SOP NAND Implementation**. I avoided using the AND OR NOT implementations because both required at least two different types of gates which would increase the complexity of the wiring diagram because I would have to have jumpers connecting the AND chip to the OR chip which may add to the visual mess of the physical build. By using only 3 NAND gates (I get the inverted output without having to NAND it) I can consolidate my work for this input on one chip keeping the layout simple.

The POS NAND implementation requires 2 NANDs to mimic the function of an AND gate so the SOP ends up being more efficient.

Requirements: 3 NAND

!K2

I'm using the **SOP and POS AND OR NOT implementation** (both SOP and POS are the same) because I'll be able to reduce the amount of chips I use if I do it this method. Because of the usage of an OR gate in JK1 I've got 3 OR gates free so using one would save me from needing to add another NAND chip. I will need to add another AND chip to the board however the !K(3 and 4) will be able to make of the unused gates.

Requirements: 1 AND, 1 OR

J3

For J3 I've decided to use the **SOP NAND implementation**. Using AND OR NOT implementation will require me to use two different kinds of chips and spread out my wiring however by using the NAND implementation I can get away with using 3 of the 4 NAND gates on the integrated circuit.

I did not use the POS NAND implementation because it would have required the use of 1 extra NAND gate.

Requirements: 3 NAND

!K3

For K3 I'm using the **SOP and POS AND OR NOT implementation** because it allows me to use the unused AND gate that was left free from the !K2 input. Both SOP and POS implementations of NAND gates requires the use of 2 NAND gates to function as one AND gate.

Requirements: 1 AND

J4

For the J input of JK4 I'm using the **SOP NAND implementation**. This allows me to use the 2 NAND gates left over from input J3 reducing the amount of chips on my board. I did not use the POS NAND implementation because it required me to use two NAND gates together to mimic the function of an AND gate while the SOP version doesn't require that of me.

Requirement: 2 NAND

!K4

I'll be using the **SOP and POS AND OR NOT implementation** for !K4 because it allows me to use one of the left over AND gates on the same chip used my !K2 and !K3. This allows me to keep my wiring neat and minimize my gate usage. The POS NAND implementation is the exact same but requires two NAND gates to mimic the function of one AND gate.

Requirements: 1 AND

Table of Implementation Summary

	J	!K
JK1	**SOP AND OR NOT Implementation** $(\overline{Q2}\,\overline{Q3}Q4) + (Q2Q3)$ Requires: 3 AND, 1 OR	**SOP and POS AND OR NOT Implementation** $\overline{Q2}Q4$ Requires: 1 AND
JK2	**SOP NAND Implementation** $\overline{(\overline{Q1Q3})(\overline{Q1Q4})}$ Requires: 3 NAND	**SOP and POS AND OR NOT Implementation** $Q1 + \overline{Q3}Q4$ Requires: 1 OR, 1 AND
JK3	**SOP NAND Implementation** $\overline{(\overline{Q1Q4})(\overline{Q1Q2})}$ Requires: 3 NAND	**SOP and POS AND OR NOT Implementation** $Q1\overline{Q2}$ Requires: 1 AND
JK4	**SOP NAND Implementation** $\overline{Q2(\overline{Q1Q3})}$ Requires: 2 NAND	**SOP and POS AND OR NOT Implementation** $\overline{Q1}Q3$ Requires: 1 AND

Table 28 Summary of implementations that will be used in the multisim and physical build sections

Summary of Chip Usage

	J1	!K1	J2	!K2	J3	!K3	J4	!K4	Total Gate Count	Total Chip Count
AND	3	1	0	1	0	1	0	1	7	2
OR	1	0	0	1	0	0	0	0	2	1
NOT	0	0	0	0	0	0	0	0	0	0
NAND	0	0	3	0	3	0	2	0	8	2

Table 29 Consolidation of how many gates and chips I'll need to implement my circuit on multisim and in the physical build

Design Comparison

Part of the rubric is comparing your design with two alternative implementations and because of that requirement, I'll include my discussion of alternative implementations here.

My goal in this sequential logic design was to strategically use a combination of AND OR NOT implementations and NAND implementations. This was because for certain J and K inputs, the AND OR NOT implementation uses the least amount of gates and for other inputs the NAND implementation uses the least amount of gates. Thankfully with my design I've been able to balance space reduction with the keeping the wiring neat and organized while also maximizing the number of gates used on each chip to minimize wastage and as such I believe that this design is the most optimal.

One could make the case that I only use 50% of the OR gates on my OR IC however if I were to use the NAND implementations in order to avoid the use of 1 OR chip, I would need to add 2 NAND chips in order fulfill the same logic.

In this section I will compare my implementation with an implementation only using AND OR NOT implementations and an implementation only using NAND. When deciding between SOP and POS I'll use the one with the least number of gates

Table of AND OR NOT Implementation Summary

	J1	!K1	J2	!K2	J3	!K3	J4	!K4	Total Gate Count	Total Chip Count
AND	3	1	2	1	1	1	1	1	11	3
OR	1	0	1	1	2	0	1	0	6	2
NOT	0	0	0	0	0	0	0	0	0	0
NAND	0	0	0	0	0	0	0	0	0	0

Table 30 Design project implementation using only AND OR NOT gates

Table of NAND Implementation Summary

	J1	!K1	J2	!K2	J3	!K3	J4	!K4	Total Gate Count	Total Chip Count
AND	0	0	0	0	0	0	0	0	0	0
OR	0	0	0	0	0	0	0	0	0	0
NOT	0	0	0	0	0	0	0	0	0	0
NAND	5	2	3	2	3	2	2	2	21	6

Table 31 Design project implementation using only NAND gates

Design Comparison Conclusion

With using only NAND gates I would need a total of 6 NAND chips added to the board, using only AND OR NOT would requires 3 AND chips and 2 OR chips, 5 in total. My current implementation also takes 5 chips so using the NAND implementation would require that I use more space than necessary. The extra space for the NAND implementation likely comes from the implementation of the !KX logic as 2 NAND gates would attempt to mimic the function of a singular AND gate.

Additionally, in my implementation, I try and make sure that the input for JX or !KX are consolidated to 1 at most 2 different chips. Unfortunately the AND OR NOT alternative implementation will have many inputs being determined by a combination of AND and OR gates. The wiring for these AND and OR chips may get complicated as each input may require logic from the AND chip and the OR chip likely causing

each flip flop's outputs and inputs mixed into different chips scattered throughout the board making debugging more complicated than necessary.

In conclusion, the AND OR NOT implementation would be the only alternative implementation that could be acceptable to adopt as it uses the same amount of gates (17 gates between AND and OR) and same number of chip space (requires 5 chips). The only difficulty is that the wiring may get messier than normal and prove to be difficult to explain in the report. The best option is to use NAND where possible as I'll be able to keep the logic to 1 chip, and then use AND and OR where it has the most efficient logic which is the exact implementation that I currently have.

Logic Optimizations and Why I Only Have One.

In the rubric Dr. Minnick mentions to state what logic optimizations you've made after K-mapping, this includes using the same logic gate output for different inputs to reduce the number of gates used.

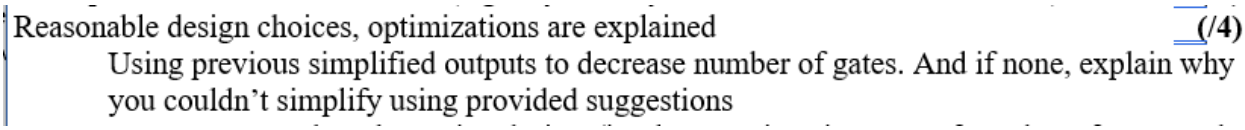
Reasonable design choices, optimizations are explained (/4)
Using previous simplified outputs to decrease number of gates. And if none, explain why you couldn't simplify using provided suggestions

Figure 1 Instructions from Design Project Tasks document describing logic optimizations

For my J4 and J2 inputs I can reuse my $\overline{\overline{Q1}Q3}$ output however unfortunately that is the only logical similarity within my circuit. The lack of crossover with logic is likely because of the small amount of "Don't care" cases I have. If more don't care cases were present, the logic would likely be less complicated making cross over more likely.

JK4	SOP NAND Implementation	JK2	SOP NAND Implementation
	$\overline{Q2(\overline{\overline{Q1}Q3})}$ Requires: 2 NAND		$\overline{(\overline{\overline{Q1}Q3})(\overline{Q1Q4})}$ Requires: 3 NAND

Figure 2 Possible circuit optimization for the design of the JK flip flop inputs J2 and J4

It is worth noting that I've made optimizations discarding the old Q4 flip flop for the ground constant and the old Q1 by replacing its output with Q2 however that is due to the nature of my student number having few distinct digits and all of them are under 8. The complexity in logic gates with this project scales proportionally with the complexity of the memory states and since I have very specific memory states, my logic is more complicated, and I have minimal overlap.

Multisim

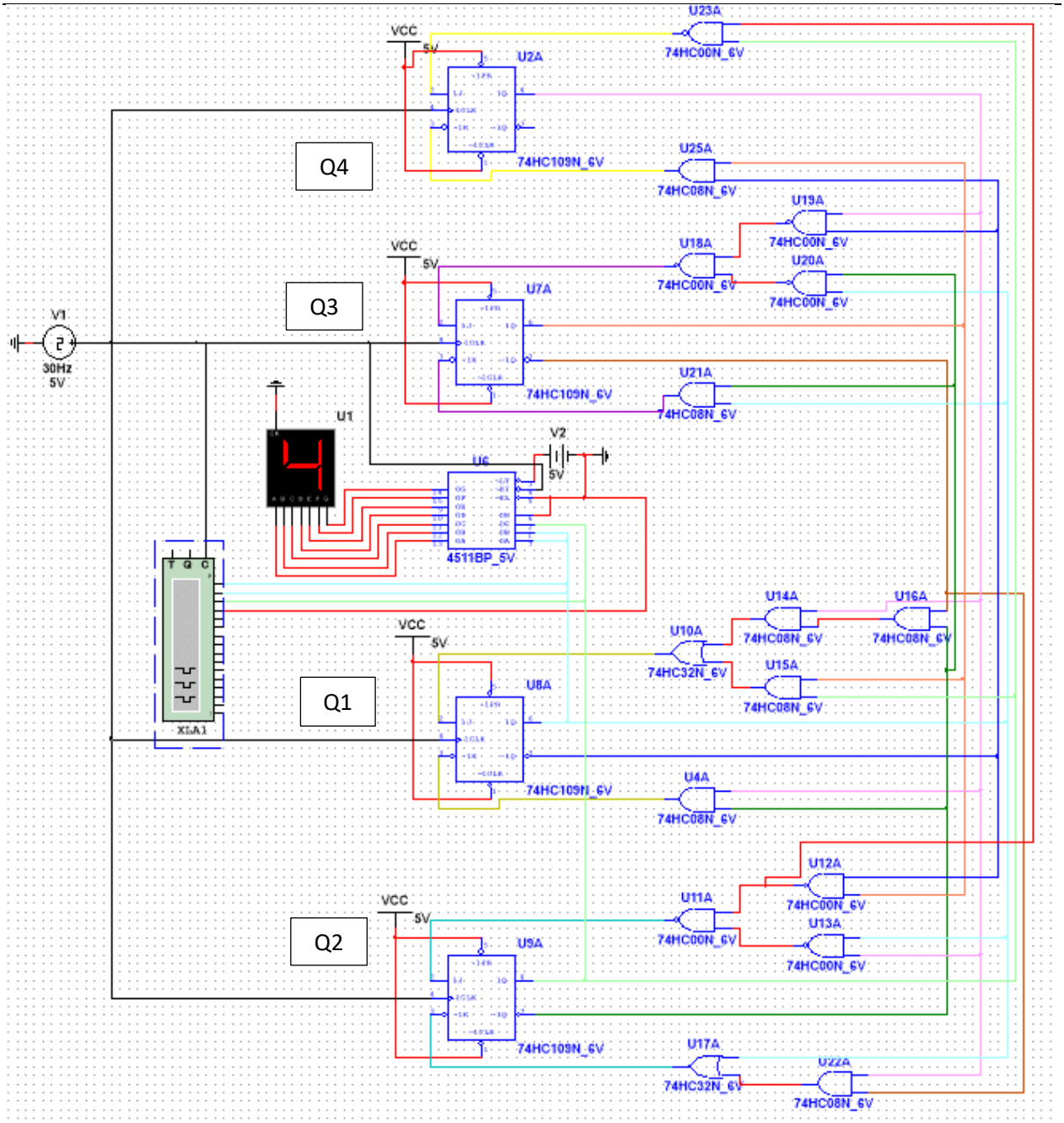

Figure 3 Multisim photo of design project

Circuit Description

Through the implementation of four flip flops and several logic gates, I have the ability to display my student number on the seven segment display with each number appearing one at a time.

In binary coded decimals below 8 there are three binary inputs that could determine what number is outputted. If I were to map out how Q1 through Q4 map onto the representation of a binary coded decimal it would look like so; Q2 Q1 Q1 (# # #) so if I wanted a 4, Q2 would output high and Q1 would output low (1 0 0). Because of the optimizations that I mentioned in the analytical section, I'm able to represent the least significant bit and the second least significant bit with Q1 and since all my numbers are below 8, my most significant bit will be represented with Q2.

Q3 and Q4 play a strong role in memory management and keeping track of which repetition of a number we're on. The outputs factor into the logic gates that connect to the J and !K inputs for each flip flop.

The outputs from Q2 and Q1 go into the BDC decoder where the binary number is converted into 7 outputs that can individually light up each segment on the 7 segment display.

It is also worth noting that the outputs of Q1 and Q2 also enter the logic analyzer to produce the timing diagram.

The inputs J and !K are controlled by a series of logic gates which get their design from the K-mapping done in the analytical section. Based off of the current outputs, the logic gates create the correct inputs to excite the JK flip flops to the correct next state.

Colour Code

	J and !K	Q1	Q2
JK1	Dark Yellow	Light Blue	Dark Blue
JK2	Teal Blue	Light Green	Dark Green
JK3	Purple	Light Orange	Dark Orange (brown)
JK4	Bright Yellow	Light Pink	Dark Pink

Table 32 Table explaining the wire colour coding on multisim for the flip flops

Additionally, black was used for wiring pertaining to the clock signal and red was used for anything connective wiring like the wiring from the BCD decoder to the 7SD display or the connections in between the logic gates.

Timing Diagram

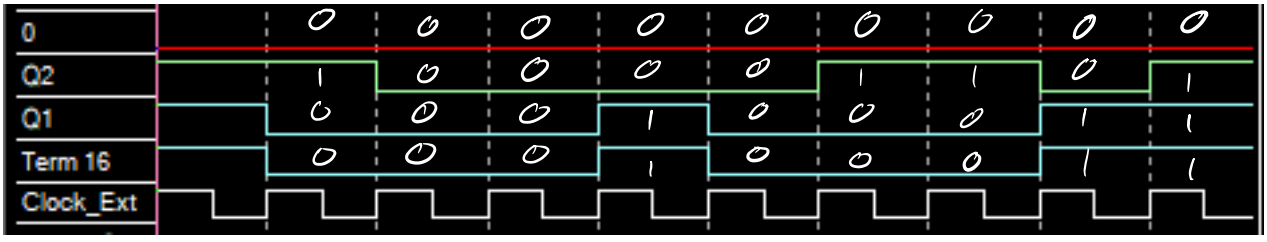

Figure 4 Timing Diagram from the multisim circuit

In this timing diagram the line labeled 0 is grounded because my student number only has numbers under 8. The green line is from the output of Q2 and both light blue lines are from the output of Q1.

We can see that the sequence follows this pattern

0100→0000→0000→0011→0000→0100→0100→0011→0111

ENGPHYS 2E04: Analog & Digital Circuits
Knighd7 || Dorian Knight || 400304437
Design Project

4 0 0 3 0 4 4 3 7

Which translates to my student number **400304437**.

Video of Simulation in Action!

https://youtu.be/Pqen74CDdvw

Physical Build

Colour Code

Colour	Purpose	
	Flip Flop breadboard	Logic gate board
Red	Q1 and !Q1	Q1 and !Q1 Connective wiring between the gates
Blue	Q2 and !Q2 and clock line	Q2 and !Q2
Green	Q3 and !Q3	Same
Yellow	Q4 and !Q4	Same
Black	J inputs for all flip flops	Same
White	!K inputs for all flip flops	Same
Orange	N/A	Connective wiring between the gates

Table 33 Colour code for the physical build

Note: Flat wires will be used to connect constants such as ground and Vcc in addition to connecting the JK flip flops to the clock line wherever possible. Since the flat wires come in various colours at different sizes, the colouring of the flat wires may not match the colour scheme.

Since white doesn't show up well on a white background, lilac was used in place of white for the design schematics.

Design Schematics

Before jumping right into the physical build it is extremely important to give an overview of the wiring to make explaining the circuit easier.

Preamble and notes on wiring diagrams

To minimize space used on the bread board, some logic gate chips will be shared between different JK flip flops. An "X" will be used to mark which logic gate in the chip has already been used for a different purpose.

Wiring: JK flip flop, Constants

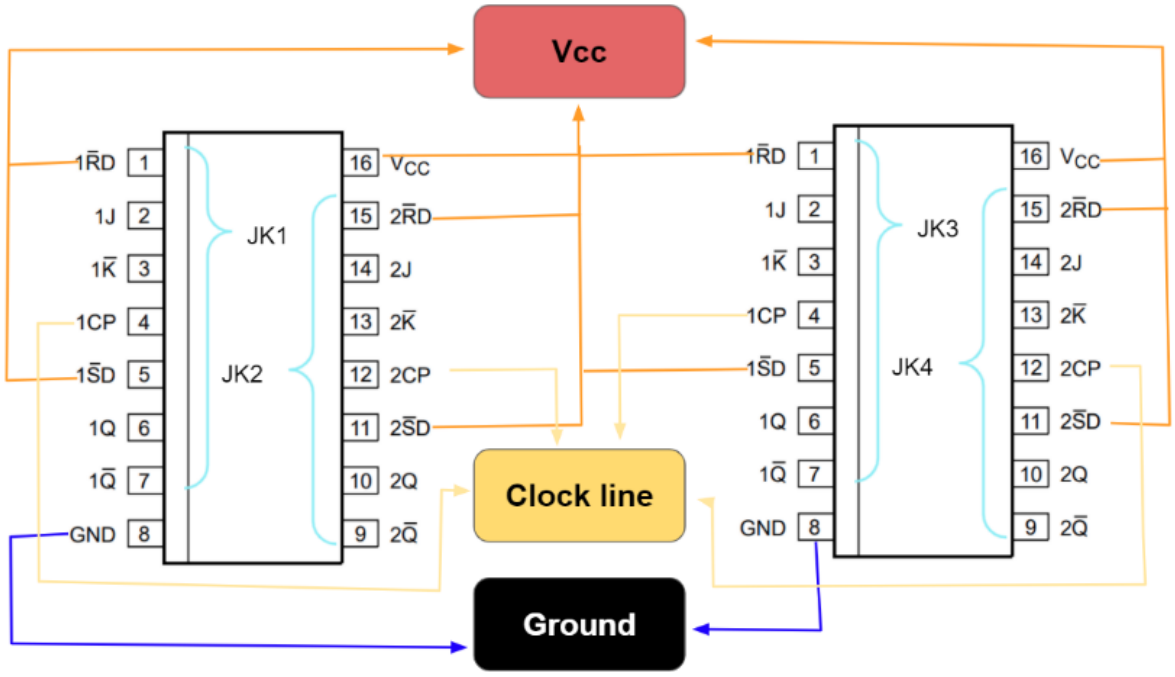

Figure 5 Wiring diagram for JK flip flop - what pins go to power, which ones to ground

The figure above shows how some of the pins of the JK flip flop will be connected to power, ground, and the clock signal coming from the hantek. Because we don't need to force any starting states in this lab, we can keep all set and preset pins held high at Vcc.

The wiring for this section will be done with flat wires to minimize the visual confusion of the circuit. The colour of these flat wires will vary so this specific diagram does not follow the aforementioned colour scheme.

Wiring: JK1 Inputs

	J	!K
JK1	**SOP AND OR NOT Implementation** $(\overline{Q2}\ \overline{Q3}Q4) + (Q2Q3)$ Requires: 3 AND, 1 OR	**SOP and POS AND OR NOT Implementation** $\overline{Q2}Q4$ Requires: 1 AND

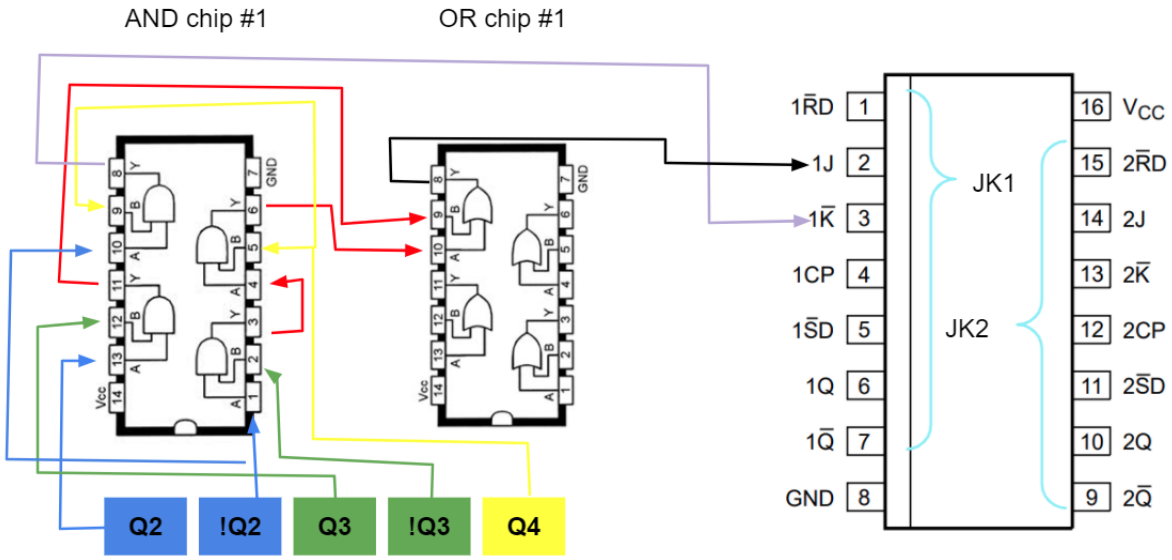

Figure 6 Wiring diagram for the inputs into JK1

Wiring: JK2 inputs

JK2	SOP NAND Implementation	SOP and POS AND OR NOT Implementation
	$\overline{(\overline{Q1Q3})(\overline{Q1Q4})}$ Requires: 3 NAND	$Q1 + \overline{Q3}Q4$ Requires: 1 OR, 1 AND

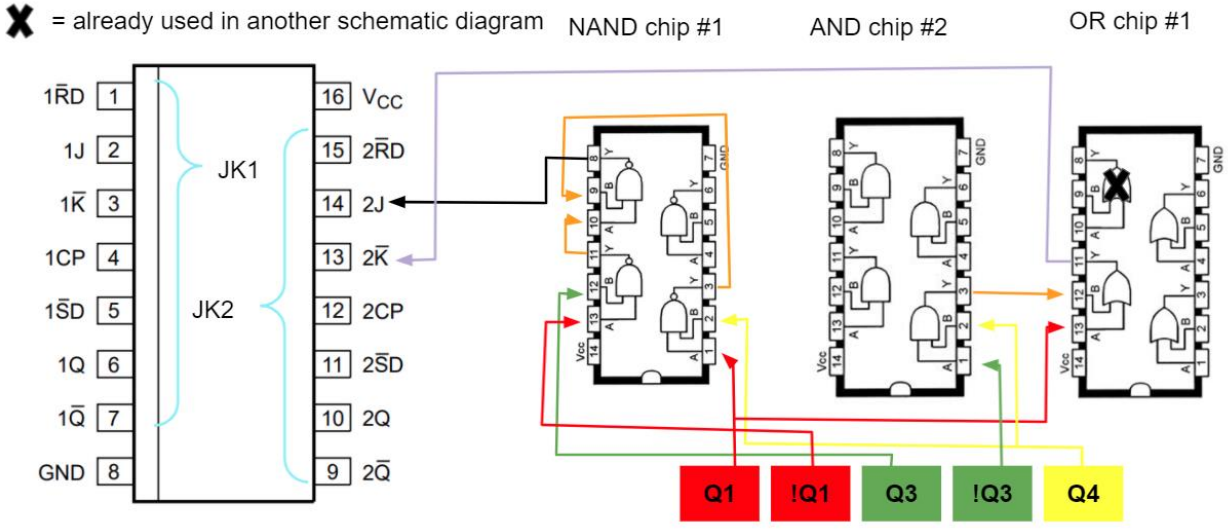

Figure 7 Wiring diagram for the inputs into JK2

JK3	SOP NAND Implementation	SOP and POS AND OR NOT Implementation
	$\overline{(\overline{Q1Q4})(\overline{Q1Q2})}$	$Q1\overline{Q2}$
	Requires: 3 NAND	Requires: 1 AND

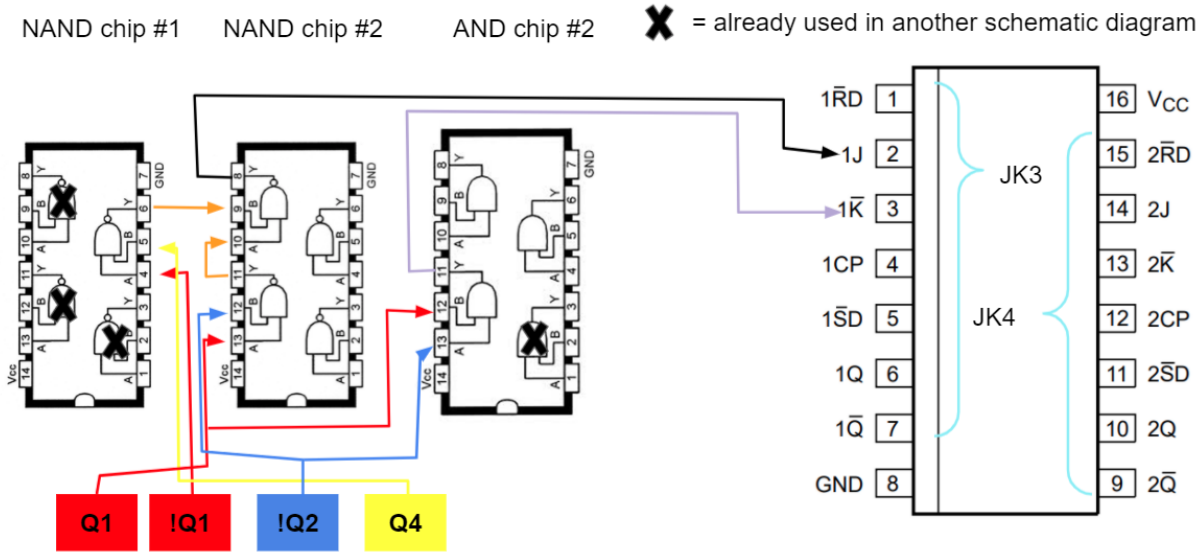

Figure 8 Wiring diagram for the inputs into JK3

JK4	SOP NAND Implementation	SOP and POS AND OR NOT Implementation
	$\overline{Q2(\overline{Q1Q3})}$	$\overline{Q1}Q3$
	Requires: 2 NAND	Requires: 1 AND

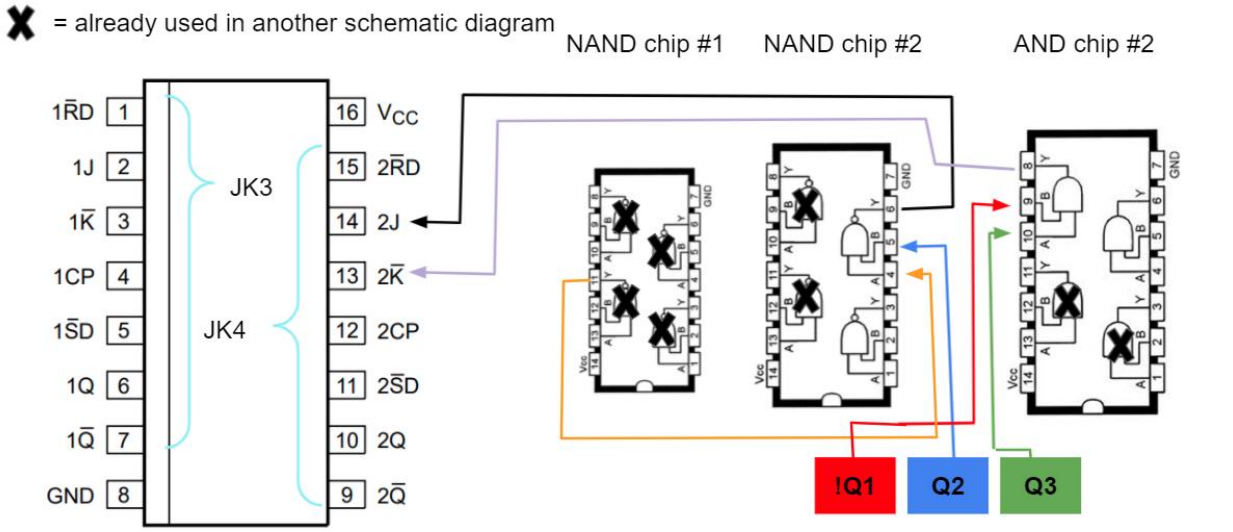

Figure 9 Wiring diagram for the inputs into JK4

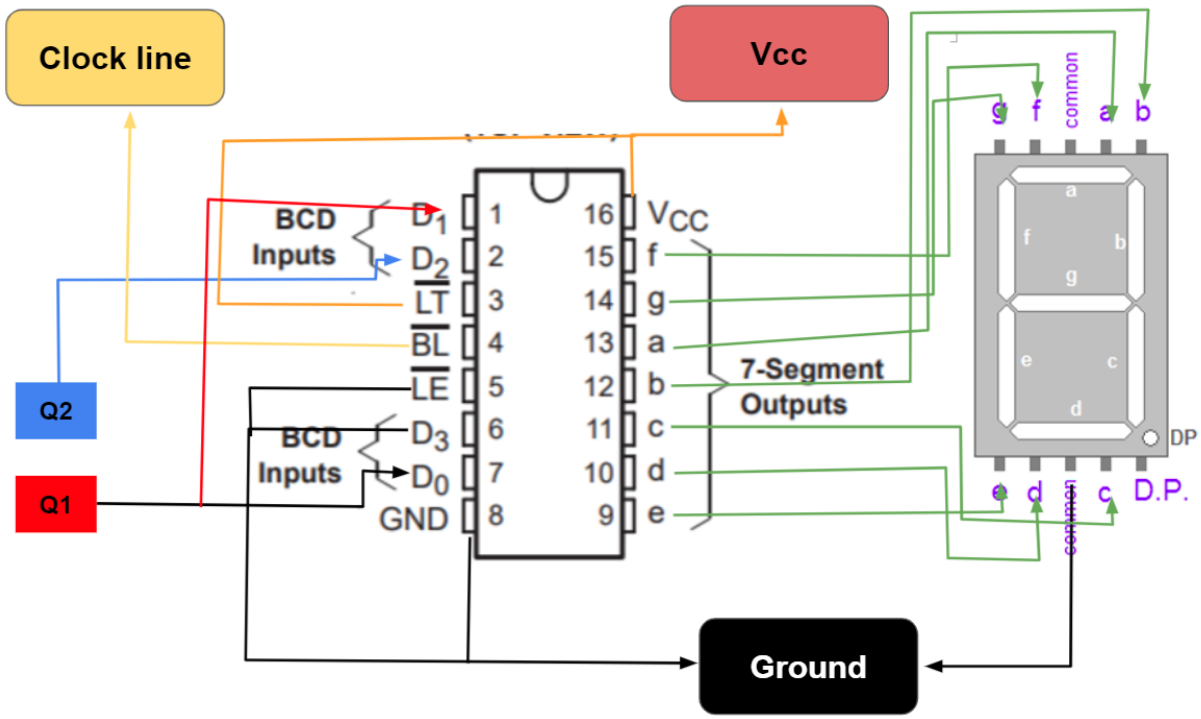

Figure 10 Wiring diagram of the BCD to 7SD decoder

The wiring to Vcc, ground, and clock lines will be accomplished using flat wires, these flat wires do not match the colour scheme. The green wiring from the BCD-7SD decoder to the 7SD display won't be green and instead the connection will be made with 10 ohm resistors.

The reason why the red Q1 box ends up having a black wire to D0 is because I ran out of wires during the physical build so to match the colour scheme, I had to use what was left.

Physical Build Process

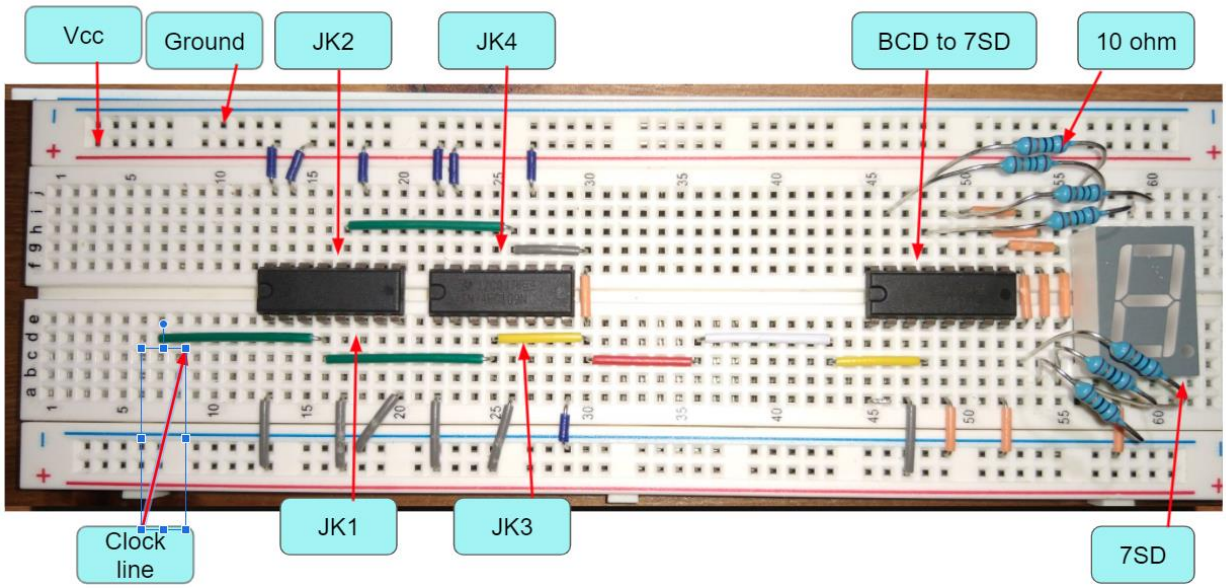

Figure 11 Initial layout of JK flip flops with BCD to 7SD decoder and 7SD – only flat wiring

I first started out by splitting my layout between the two breadboards, I decided to use one board for the JK flip flops and the 7SD portion and then use the other breadboard for all of the logic gates.

This photo here shows all of the flat wiring layout. The purpose of using the flat wires is to minimize visual confusion when I use the jumper wires to connect inputs and outputs (J, !K, Q, !Q). The flat wires go to anything that is essential for the chip to operate, this includes Vcc, ground, connecting the clocks together and even wiring the preset and clear pins to Vcc. The wires running vertically hook !preset, !clear and Vcc to Vcc in addition to grounding the pins to ground. The wires that run horizontally synchronize all of the clocks for each flip flop such that I can hook in the clock signal to any one of those wires and all of the flip flops will get the same clock pulse.

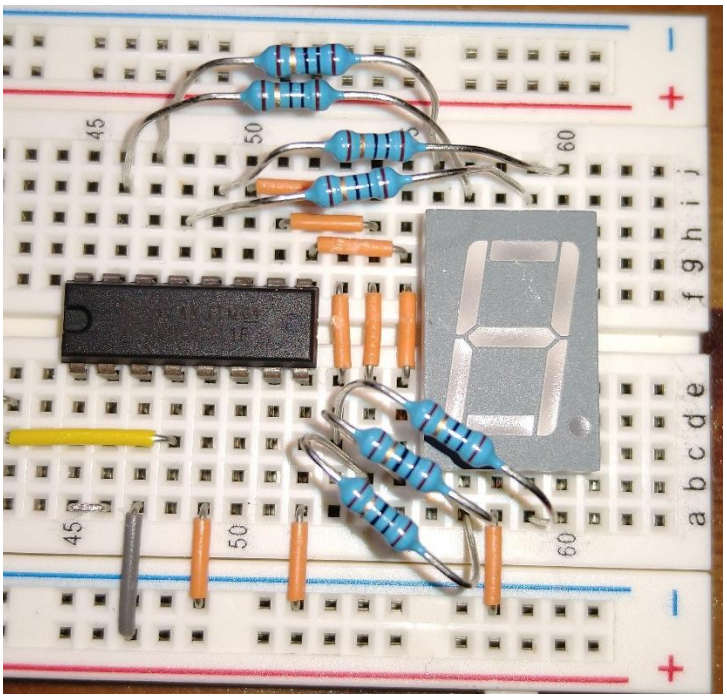

Figure 12 Close up of 7SD region of the breadboard

Next up, I've included a close up of the 7SD portion of the breadboard where I've extended the clock line from the JK flip flops to connect with the !BL pin. I've also got the !LT pin hooked up to Vcc and !LE to ground to enable blinking when the number changes.

I also have 10 ohm resisters branching out from the outputs that will then output into the 7SD to display my student number. The 7SD is connected to ground with the orange flat wire.

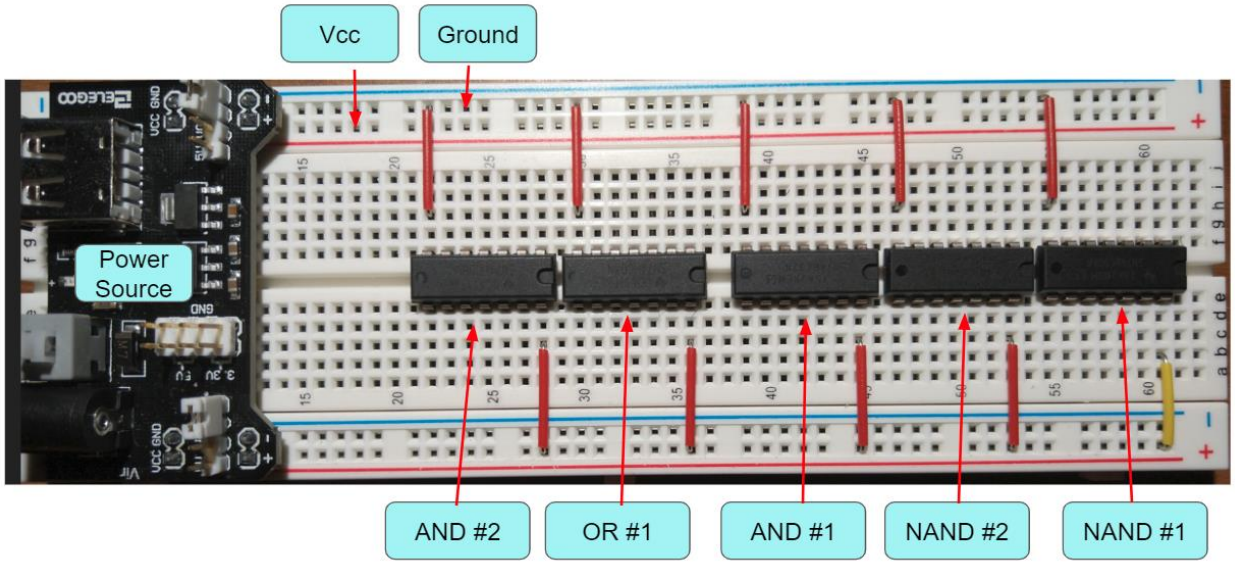

Figure 13 Logic gate layout on auxiliary breadboard

While I have the JK and 7SD chips on the first breadboard, I've moved the logic gates to the second breadboard in the efforts of spacing things out. I've purposely put the OR chip in between both And chips because both And chips will need to use an OR gates as part of the logic leading back to the JK flip flops. All of the pins connect to Vcc and ground on either side of the breadboard and the Elegoo header board supplies both rails with 3.3 V.

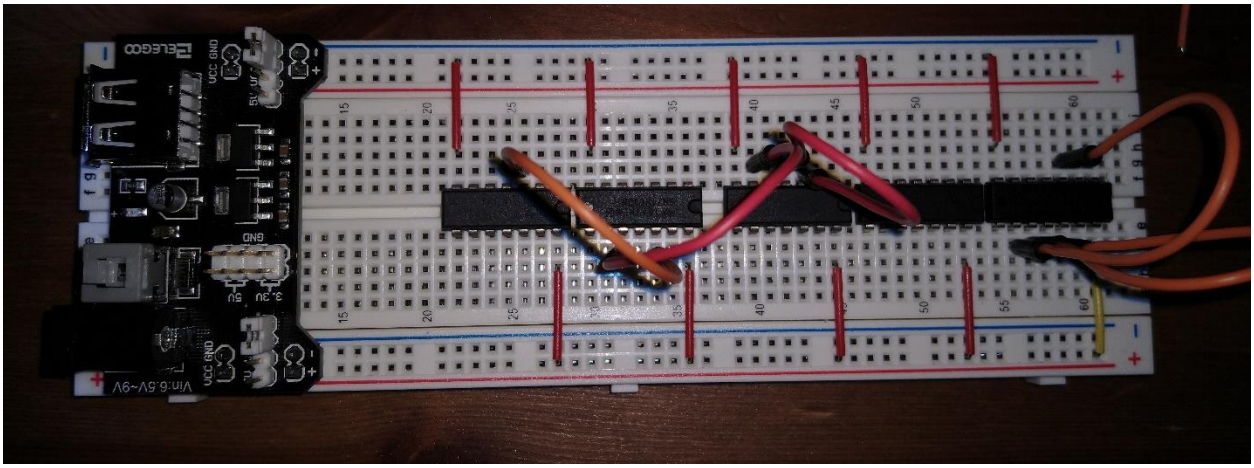

Figure 14 Intra-chip wiring of the logic gate breadboard

Starting off the wiring I decided to commence with any wires that connect the output of one logic gate to the input of another. For the outputs of AND gate #1 I've used red wiring (because I've run out of orange) and for all of the other logic gates I've used orange wires. The next step is to start connecting the outputs of each flip flop to the required inputs on the logic chips.

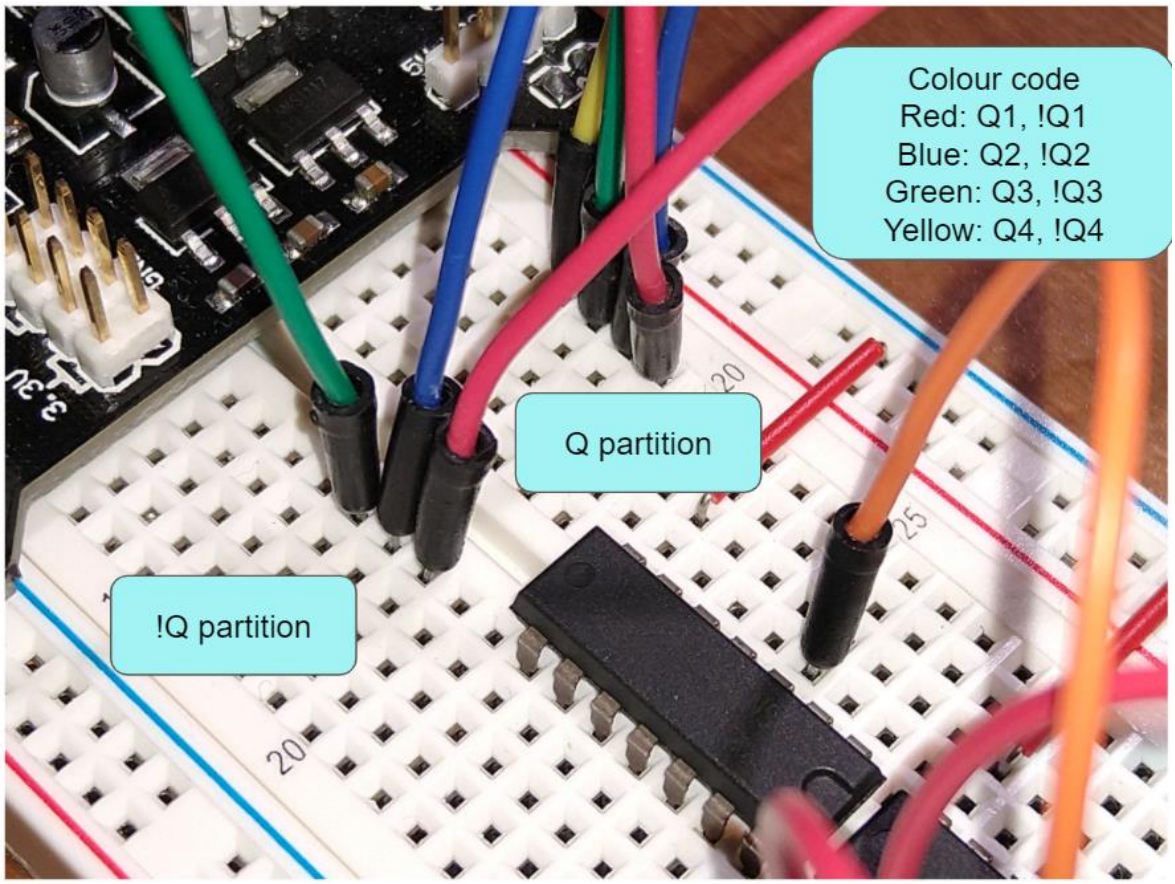

Figure 15 Outputs from the flip flops will arrive onto the logic gate breadboard and propagte throughout the row

To make the wiring easy to debug, instead of having every logic gate input that requires the output of a flip flop (eg !Q2, or Q1) plug into the other breadboard, I will keep the wiring localized on the bread board by plugging in all of the Q1 inputs into line 19 on the right side, and all Q2 inputs on line 18 on the right side. The same goes for all of the !Qx inputs as well however they will go to the left side of the breadboard. Please note that there is no line allocated for !Q4 because my logic doesn't depend on the state of !Q4.

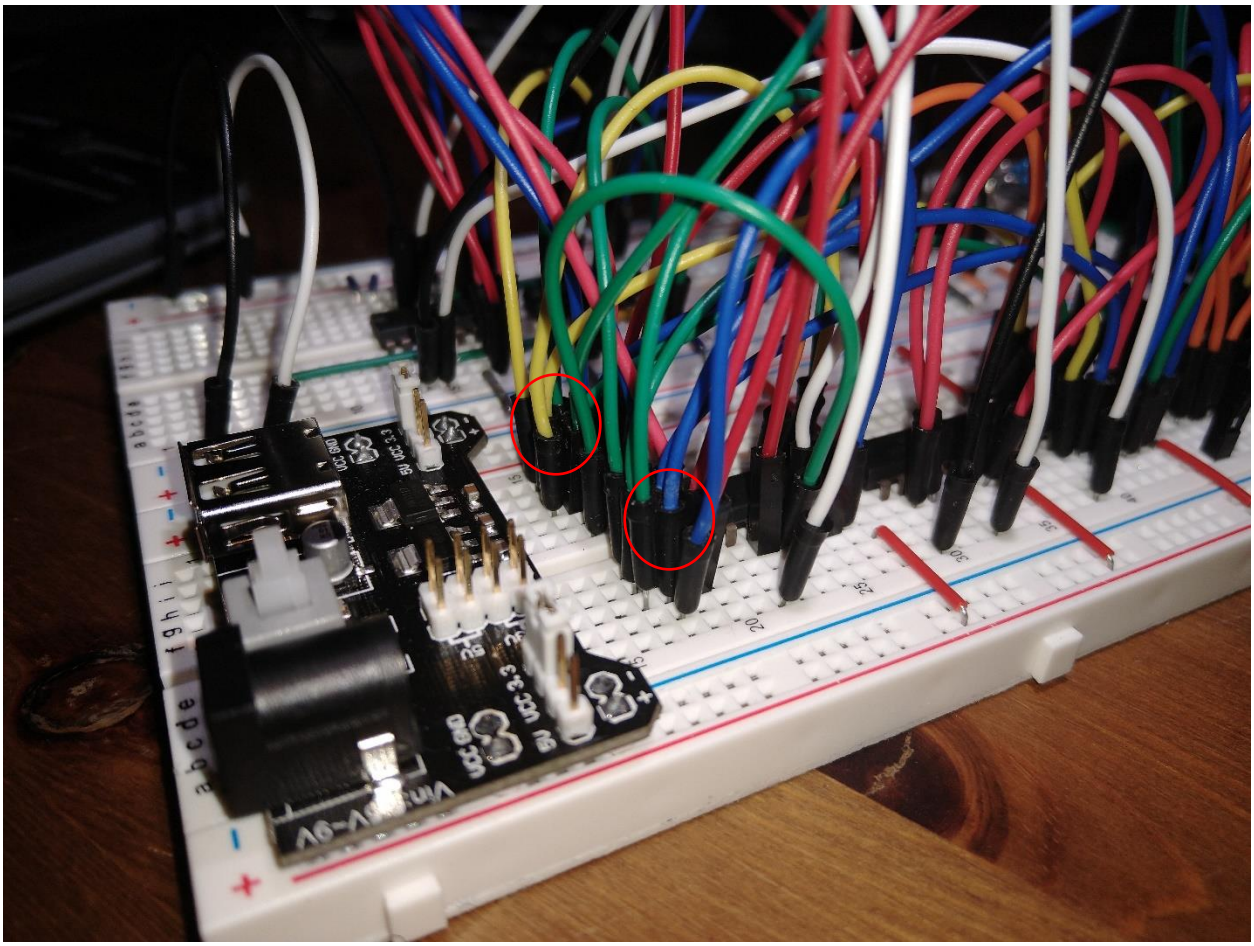

Figure 16 Implementation of the !Q and Q partition on the logic gate breadboard

Now that all the wiring has been implemented, the !Q and Q partitions helped to organize the breadboard as well as debug. I had a few Q3 outputs plugged into !Q3 by accident and was able to spot it due to the organized structure. In this photo the !Qs are on the right and the Qs are on the left.

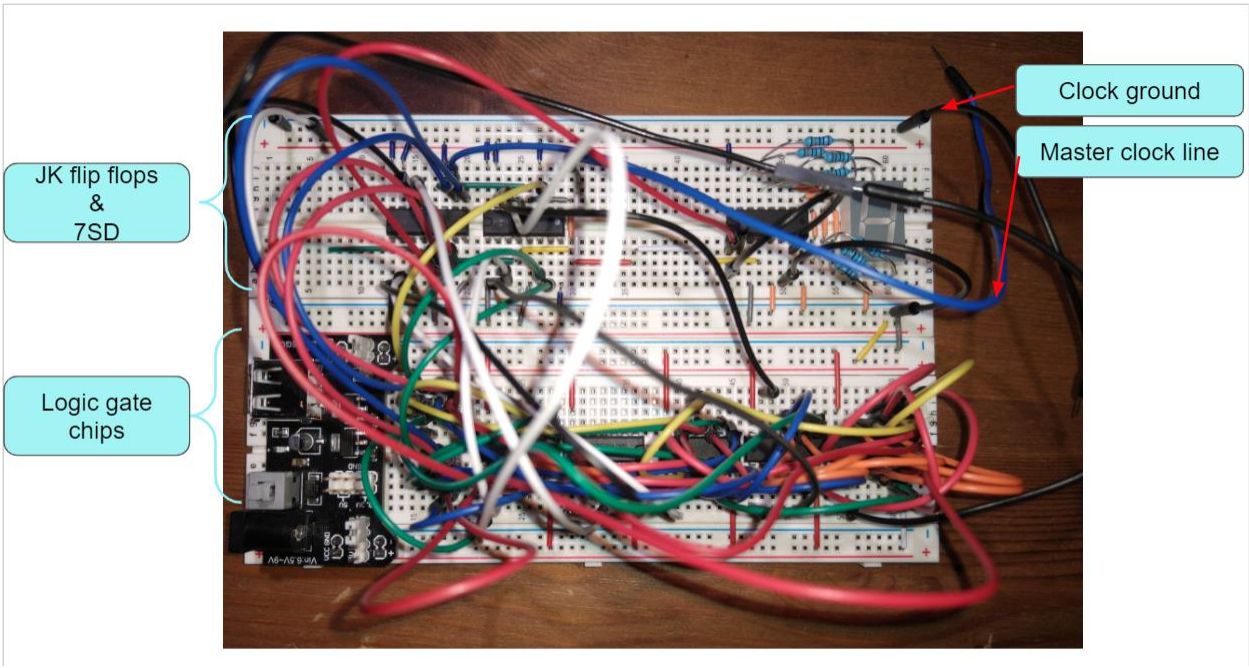

Figure 17 Completed circuit with all necessary wiring

The photo above is of my circuit with all of the wiring completed with the logic gates on the southern most breadboard and the flip flops and 7SD portion on the northern most breadboard. I've also added labels describing the clock power and ground which is used to synchronously drive the entire circuit using the network of flat wires I assembled previously.

All of the added wiring is used to link the inputs of the logic gates with the respective outputs from the flip flops and the white and black wires coming from the outputs of the logic gates loop back to the flip flops serving as inputs for J and !K.

Because the wiring is a bit intense I will upload a couple of photos from slightly different angles.

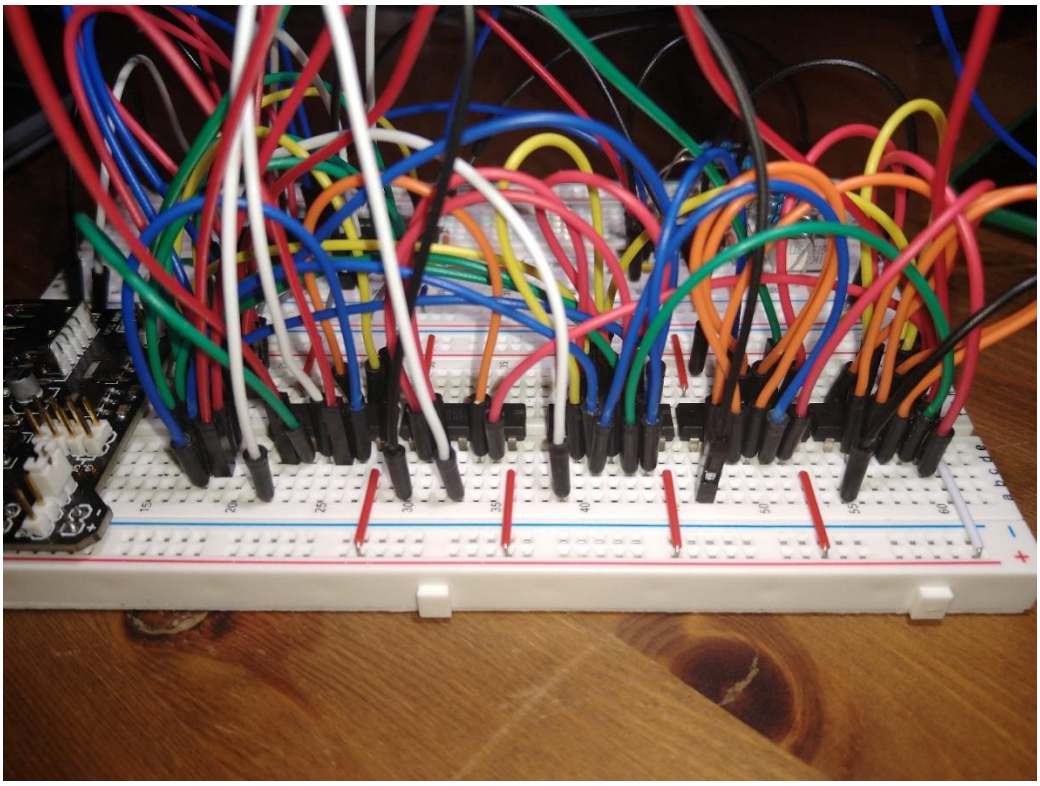

Figure 18 Completed circuit - front angle. View of logic gate breadboard and the forest of wires

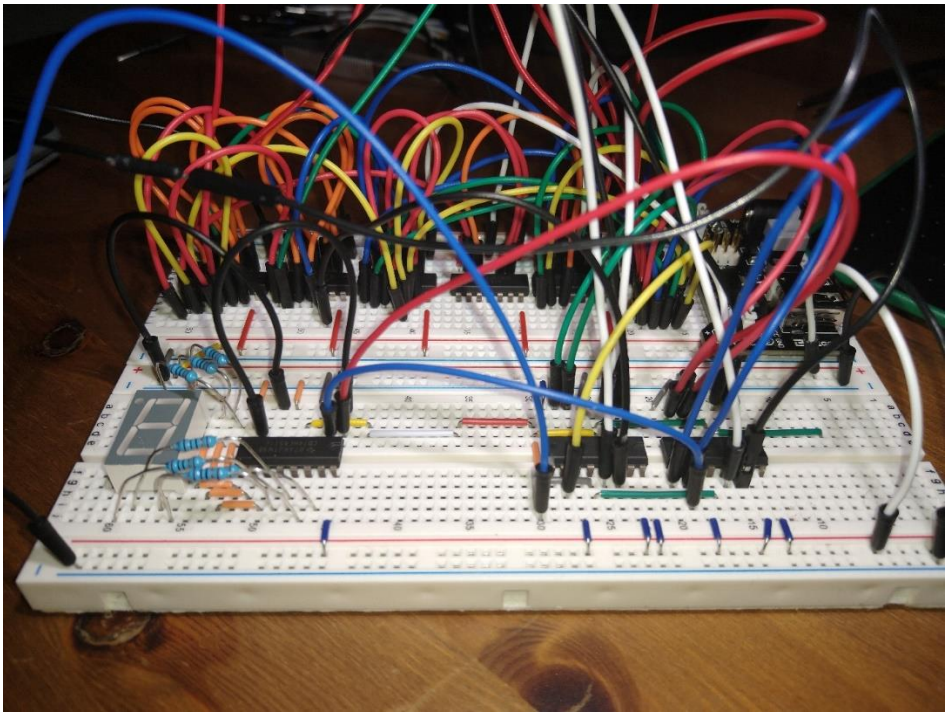

Figure 19 Completed circuit - back angle. View of JK flip flop inputs and outputs in addition to the 7SD implementation

It is also worth noting here that the power and ground for this segment of the breadboard comes from flat wire connections coming from the breadboard with the Elegoo headboard.

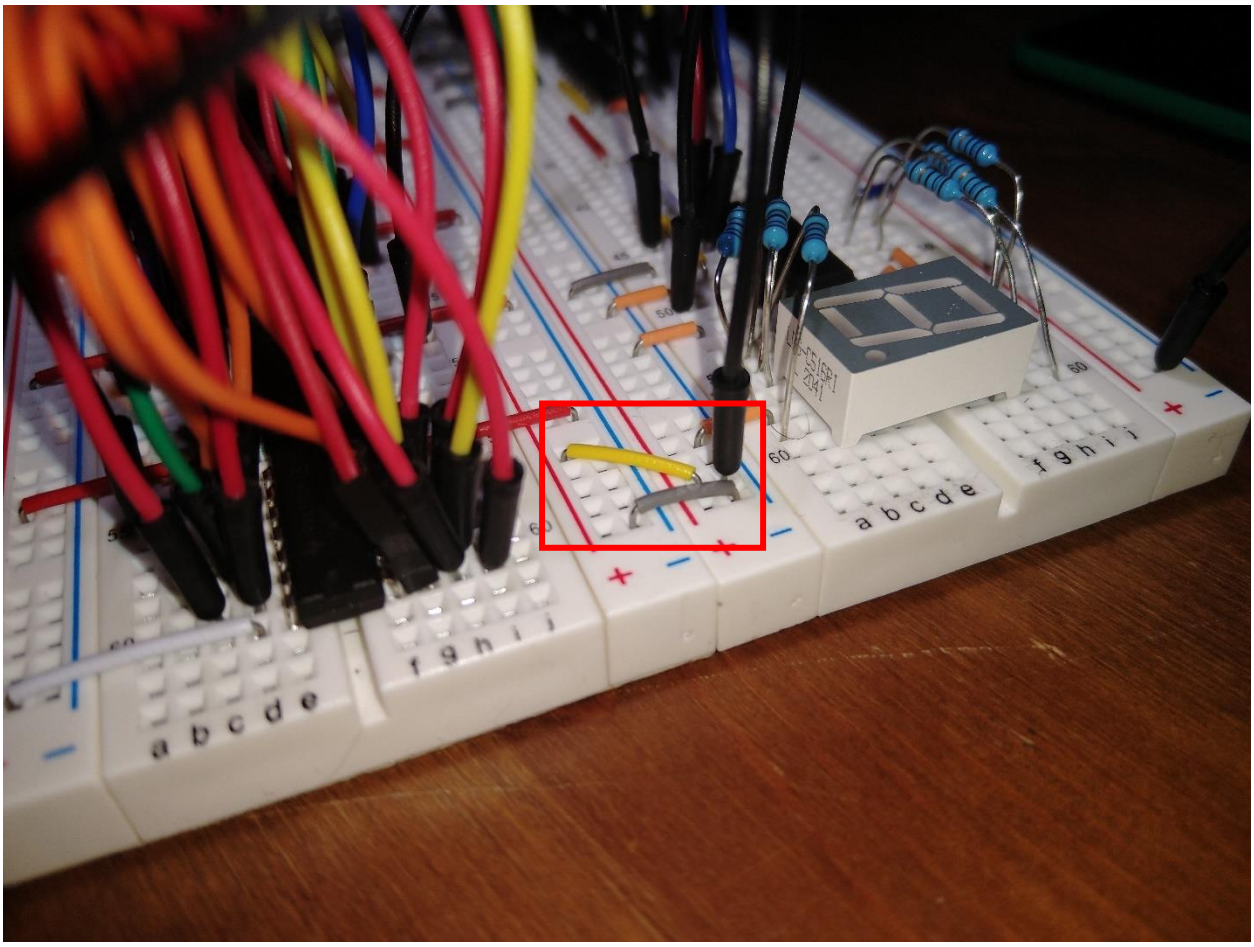

Figure 20 Power and ground connection between the two breadboards

Last but not least I realized I forgot to take a photo of the board with my student card in the frame so here is another top down view of the circuit – this time with my student card

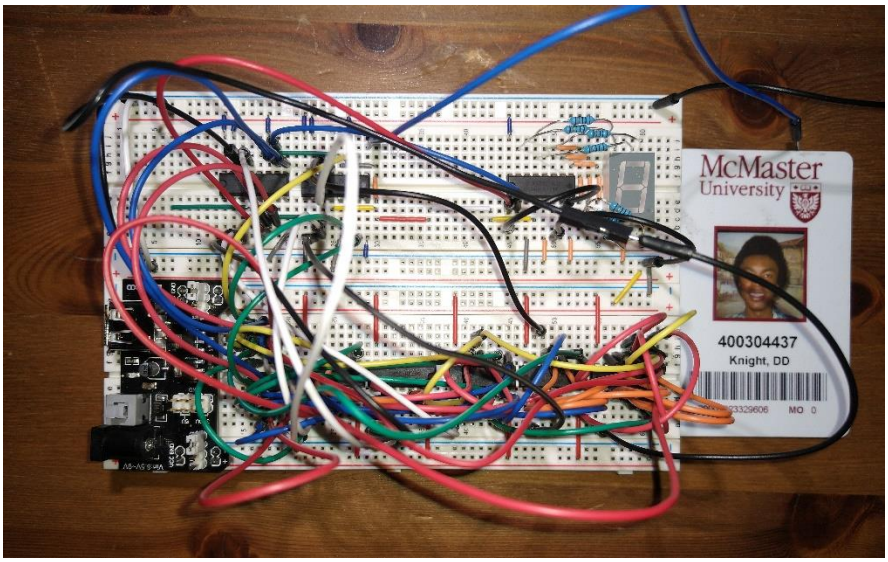

Figure 21 Top down view of circuit board with student card in frame

Debugging

At first my circuit didn't work as expected, only cycling through numbers 4,3,0 or 4,3,7 on repeat. From there I looked at the logic gate wiring into the specific gates in comparison to my schematic and everything looked up to par. I then compared my schematic to my multisim and analytical work and everything also seemed up to par. Only when I looked at my connection between the logic gate inputs and the !Qx/Qx bus higher up on the breadboard did I notice that I had 2 wires that were supposed to connect to Q3 instead inserted into the !Q3 line. I promptly fixed that error and the implementation worked exactly as demonstrated in the video below

Circuit Video

Circuit in action: https://youtu.be/bVT8jeTuHBg

Circuit explanation: https://youtu.be/0QYvIMZ_dyE

Conclusion

In the end, the logic design from the analytical section when implemented in multisim and the physical build have been shown to work as expected and output my student number.

No bugs or errors are present in the final build which leads me to believe that my analytical section (k-mapping, truth tables, excitation tables) was correct.

Video Component

For the final design project, us student must create a video explaining our report – you can find mine here below.

https://youtu.be/ttw1BfveHq8

Appendix

Integrated Circuit Pinouts

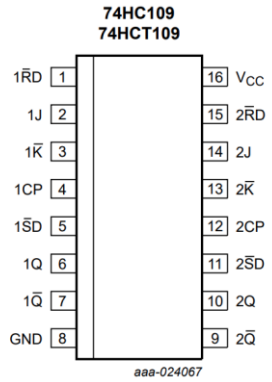

Figure 22 74HC109 JK Flip Flop

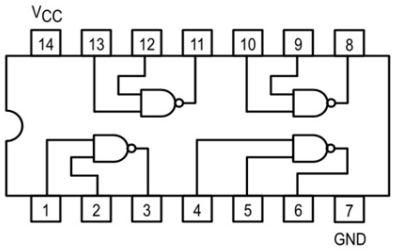

Figure 23 74HC00 NAND

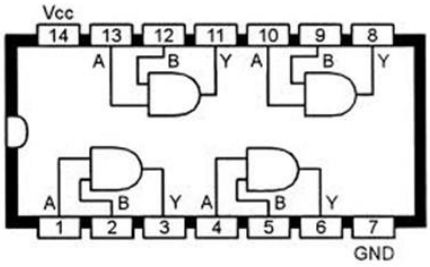

Figure 24 74HC08 AND

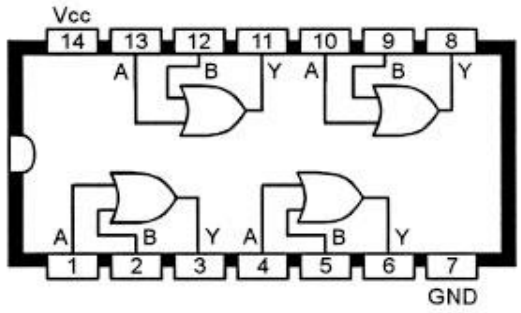

Figure 25 74HC32 OR

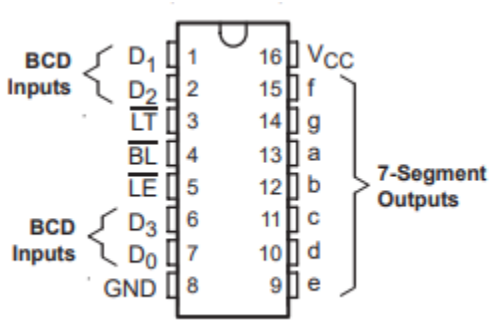

Figure 26 CD74HC4511E BCD to 7SD