Sequential Logic Design Project – Seven Segment Display

Wilson Liu

Dec 6, 2022

DESIGN PROJECT TASK:

Get a 7-segment display (7SD) to cycle through your student number by using **sequential logic design.** Show your logic synthesis, K-mapping, optimizations for number of gates used, simulation in NI Multisim, and lastly build and physically implement this design using only logic gate ICs, wires, and a 7SD chip.

ANALYTICAL:

My student number is 400330928. First, we write out each of the numbers in BCD.

#	q1	q2	q3	q4
4	0	1	0	0
0	0	0	0	0
0	0	0	0	0
3	0	0	1	1
3	0	0	1	1
0	0	0	0	0
9	1	0	0	1
2	0	0	1	0
8	1	0	0	0

However, as there are multiple repeated digits (0, 3), counter flip-flops are required to implement this sequence. These flip-flops give each state a unique state identifier – which will be key in designing the FSM.

#	q1	q2	q3	q4	c1	c2
4	0	1	0	0	X	X
0	0	0	0	0	0	0
0	0	0	0	0	0	1
3	0	0	1	1	0	X
3	0	0	1	1	1	X
0	0	0	0	0	1	X
9	1	0	0	1	X	X
2	0	0	1	0	X	X
8	1	0	0	0	X	X

One possible optimization is to choose the counter values to always toggle between 0 and 1.

This would allow the J & K input columns for these counters to either always be 0 or always be 1, eliminating the need for any gates. This is due to the nature of the state transition table.

Unfortunately, that's not possible here as there are an odd number of rows – since a McMaster student number is made up of 9 digits.

The next best alternative is to make them toggle as much as possible. Specifically with only either repeated 00s or repeated 11s in each column.

This will lead to one of the input columns (either J or K) to require 0 gates. The other column may be much simpler to implement as well, if the number of repeated 00s/11s is minimized.

This can be done by noting that it is arbitrary which 3 is assigned a 1 or 0 for the c1 bit, and swapping them around to more closely fit our intended design.

#	q1	q2	q3	q4	c1	c2
4	0	1	0	0	X	X
0	0	0	0	0	0	0
0	0	0	0	0	0	1
3	0	0	1	1	1	X
3	0	0	1	1	0	X
0	0	0	0	0	1	X
9	1	0	0	1	X	X
2	0	0	1	0	X	X
8	1	0	0	0	X	X

Then the rest of the Xs can be replaced by alternating 1s and 0s, creating counters that fit our intended design.

#	q1	q2	q3	q4	c1	c2
4	0	1	0	0	1	1
0	0	0	0	0	0	0
0	0	0	0	0	0	1
3	0	0	1	1	1	0
3	0	0	1	1	0	1
0	0	0	0	0	1	0
9	1	0	0	1	0	1
2	0	0	1	0	1	0
8	1	0	0	0	0	1

Note that given the odd number of rows, the best design we can hope for will have only one instance of a repeated 00 or 11. This is accomplished with the table above.

Truth Table:

Next, we will determine the inputs for each flip-flop to achieve the required state transitions – and create this **truth table.**

#	q1	q2	q3	q4	c1	c2	**J1**	!K1	**J2**	!K2	J3	!K3	**J4**	!K4	JC1	!KC1	JC2	!KC2
4	0	1	0	0	1	1	**0**	X	**X**	0	**0**	X	**0**	X	X	0	X	0
0	0	0	0	0	0	0	**0**	X	**0**	X	**0**	X	**0**	X	0	X	1	X
0	0	0	0	0	0	1	**0**	X	**0**	X	**1**	X	**1**	X	1	X	X	0
3	0	0	1	1	1	0	**0**	X	**0**	X	**X**	1	**X**	1	X	0	1	X
3	0	0	1	1	0	1	**0**	X	**0**	X	**X**	0	**X**	0	1	X	X	0
0	0	0	0	0	1	0	**1**	X	**0**	X	**0**	X	**1**	X	X	0	1	X
9	1	0	0	1	0	1	**X**	0	**0**	X	**1**	X	**X**	0	1	X	X	0
2	0	0	1	0	1	0	**1**	X	**0**	X	**X**	0	**0**	X	X	0	1	X
8	1	0	0	0	0	1	**X**	0	**1**	X	0	X	**0**	X	1	X	X	1

This is done using a standard JK flip-flop excitation table.

Q_n	Q_{n+1}	J_n	$\sim K_n$
0	0	0	X
0	1	1	X
1	0	X	0
1	1	X	1

The don't cares in the input columns are a result of the state transitions & excitation table.

From this table, we can immediately recognize that:

!K1 = 0

!K2 = 0

!KC1 = 0

JC2 = 1

We can also take a look at if any of the inputs matches one of the output columns.

c1	!K4
1	X
0	X
0	X
1	1
0	0
1	X
0	0
1	X
0	X

c2	JC1
1	X
0	0
1	1
0	X
1	1
0	X
1	1
0	X
1	1

!K4 = c1

JC1 = c2

K-Maps:

Next, we will make the K-maps for the remaining inputs. These will be 6-input K-maps.

J1		c2 = 0				c2 = 1			
	q1q2\q3q4	00	01	11	10	00	01	11	10
c1 = 0	00	0	X	X	X	0	X	0	X
	01	X	X	X	X	X	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	X	X	X	X
c1 = 1	00	1	X	0	1	X	X	X	X
	01	X	X	X	X	0	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	X	X	X	X

Unfortunately, the boxes may be hard to read due to the nature of these 3D K-maps. For example, this is one size 8 box that wraps around the sides of the bottom-left K-map. However, they can be interpreted from the Boolean expression that follows.

J1 = !q4*c1*!c2

J2		c2 = 0				c2 = 1			
	q1q2\q3q4	00	01	11	10	00	01	11	10
c1 = 0	00	0	X	X	X	0	X	0	X
	01	X	X	X	X	X	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	1	0	X	X
c1 = 1	00	0	X	0	0	X	X	X	X
	01	X	X	X	X	X	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	X	X	X	X

One size 16 box.

J2 = q1*!q4

J3		c2 = 0				c2 = 1			
	q1q2\q3q4	00	01	11	10	00	01	11	10
c1 = 0	00	0	X	X	X	1	X	X	X
	01	X	X	X	X	X	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	0	1	X	X
c1 = 1	00	0	X	X	X	X	X	X	X
	01	X	X	X	X	0	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	X	X	X	X

One size 32 box, one size 8 box.

J3 = c2*!q1*!q2 + q4

!K3		c2 = 0				c2 = 1			
	q1q2\q3q4	00	01	11	10	00	01	11	10
c1 = 0	00	X	X	X	X	X	X	0	X
	01	X	X	X	X	X	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	X	X	X	X
c1 = 1	00	X	X	1	0	X	X	X	X
	01	X	X	X	X	X	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	X	X	X	X

One size 16 box.

!K3 = !c2*q4

J4		c2 = 0				c2 = 1			
	q1q2\q3q4	00	01	11	10	00	01	11	10
c1 = 0	00	0	X	X	X	1	X	X	X
	01	X	X	X	X	X	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	0	X	X	X
c1 = 1	00	1	X	X	0	X	X	X	X
	01	X	X	X	X	0	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	X	X	X	X

Two different size 8 boxes.

J4 = c1*!c2*!q3 + c2*!c1*!q1

!KC2		c2 = 0				c2 = 1			
	q1q2\q3q4	00	01	11	10	00	01	11	10
c1 = 0	00	X	X	X	X	0	X	0	X
	01	X	X	X	X	X	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	1	0	X	X
c1 = 1	00	X	X	X	X	X	X	X	X
	01	X	X	X	X	0	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	X	X	X	X

One size 16 box.

!KC2 = q1*!q4

Results of the K-mapping:

Input	Boolean Expression	Gates
J1	!q4*c1*!c2	2 ANDs
!K1	0	
J2	q1*!q4	1 AND
!K2	0	
J3	c2*!q1*!q2 + q4	2 ANDs, 1 OR
!K3	!c2*q4	1 AND
J4	c1*!c2*!q3 + c2*!c1*!q1	4 ANDs, 1 OR
!K4	c1	
JC1	c2	
!KC1	0	
JC2	1	
!KC2	q1*!q4	1 AND

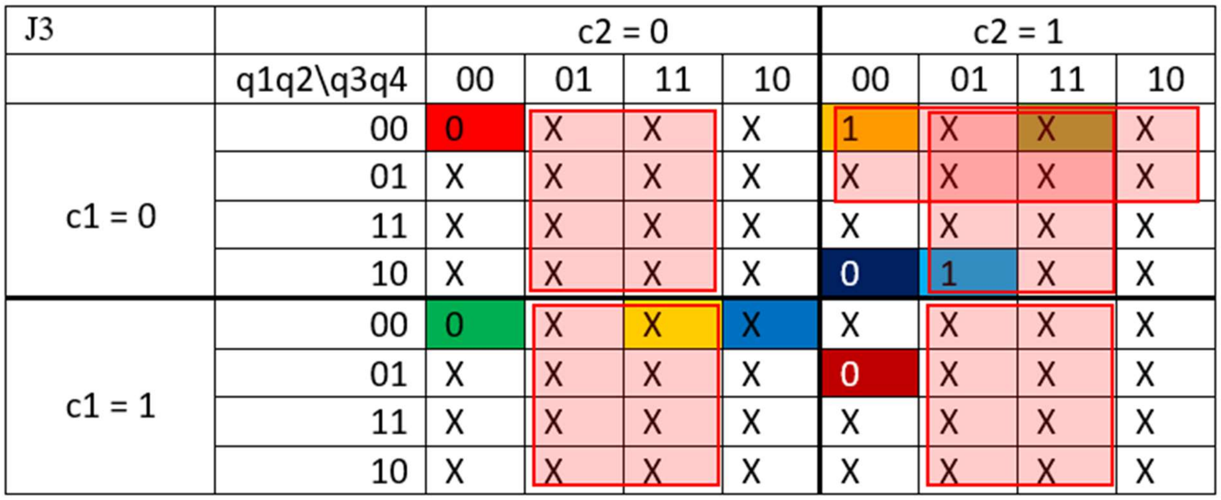
This is the initial implementation we came up with. It requires a total of 11 ANDs & 2 ORs – for a total of 13 gates & 4 chips (there are 4 gates of the same type to a chip).

Note that none of these expressions can be further simplified by Boolean algebra.

Optimization 1: Alternative Simplified SOP Expressions

As I was doing the K-mapping, I noticed there were several **alternative simplified SOP expressions** that could have also been drawn. Here, we explore those and see if they can lead to any reduction in the number of gates.

An alternative K-map for J3 is this:



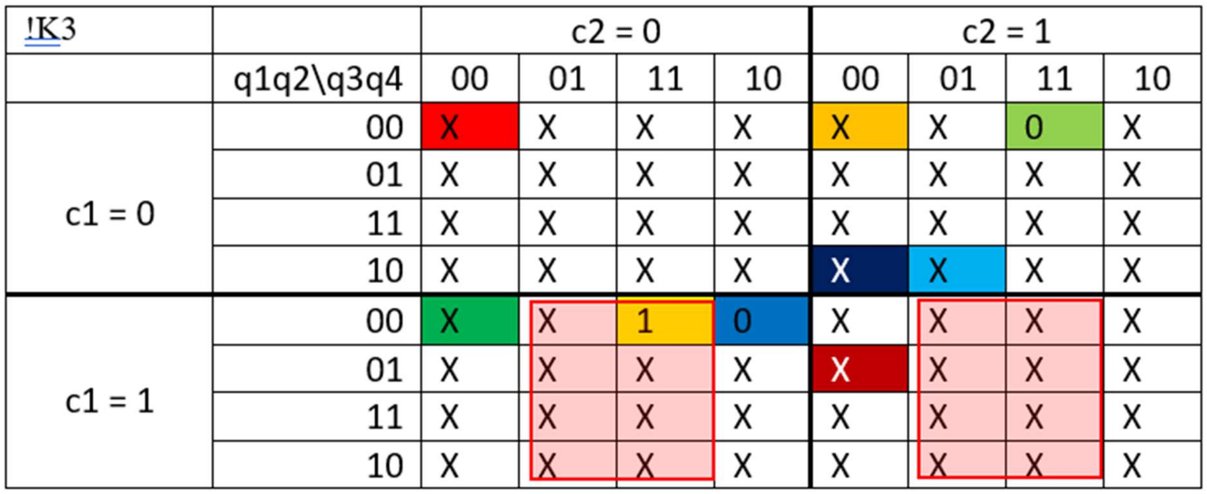
J3		c2 = 0				c2 = 1			
	q1q2\q3q4	00	01	11	10	00	01	11	10
c1 = 0	00	0	X	X	X	1	X	X	X
	01	X	X	X	X	X	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	0	1	X	X
c1 = 1	00	0	X	X	X	X	X	X	X
	01	X	X	X	X	0	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	X	X	X	X

J3 = q4 + c2*!c1*!q1 -> 1 OR, 2 ANDs

However, this leads to the same number of gates as the previously derived expression.

J3 = c2*!q1*!q2 + q4 -> 1 OR, 2 ANDs

An alternative K-map for !K3 is this:



!K3		c2 = 0				c2 = 1			
	q1q2\q3q4	00	01	11	10	00	01	11	10
c1 = 0	00	X	X	X	X	X	X	0	X
	01	X	X	X	X	X	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	X	X	X	X
c1 = 1	00	X	X	1	0	X	X	X	X
	01	X	X	X	X	X	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	X	X	X	X

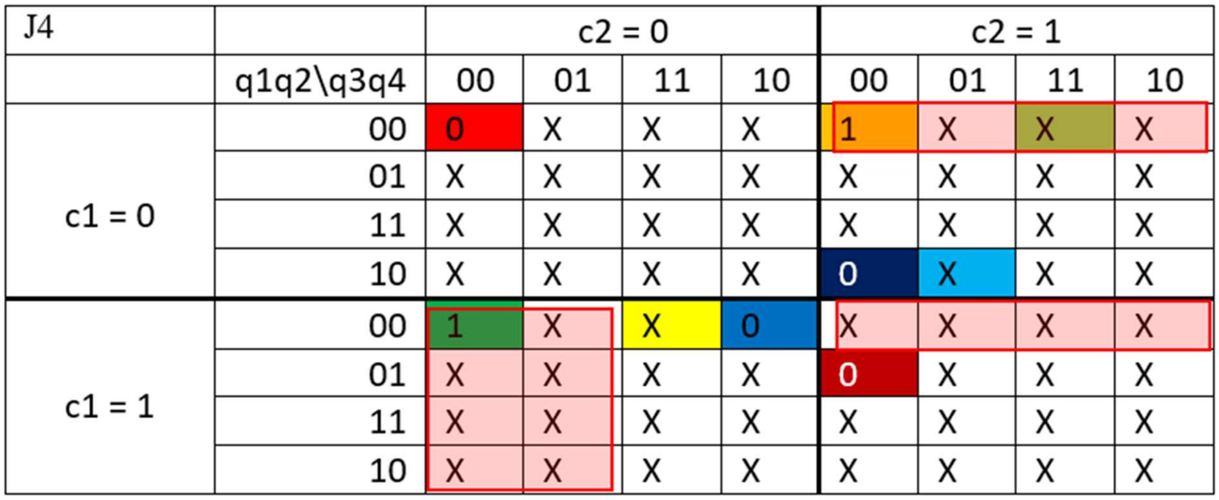
!K3 = c1*q4 -> 1 AND

This also leads to the same number of gates though.

!K3 = !c2*q4 -> 1 AND

It can be noted that there would have been a difference in the number of gates during Lab 6 (digital logic design). The original K-map would have required one more inverter. However, as these flip-flops have both Q & !Q output pins, there is no difference in gates between the two.

An alternative K-map for J4 is this:



J4		c2 = 0				c2 = 1			
	q1q2\q3q4	00	01	11	10	00	01	11	10
c1 = 0	00	0	X	X	X	1	X	X	X
	01	X	X	X	X	X	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	0	X	X	X
c1 = 1	00	1	X	X	0	X	X	X	X
	01	X	X	X	X	0	X	X	X
	11	X	X	X	X	X	X	X	X
	10	X	X	X	X	X	X	X	X

J4 = c1*!c2*!q3 + c2*!q1*!q2 -> 1 OR, 4 ANDs

Again, this leads to the same number of gates as before.

J4 = c1*!c2*!q3 + c2*!c1*!q1 -> 1 OR, 4 ANDs

Unfortunately, this optimization did not lead to any change in the number of gates. The implementation still uses a total of 11 ANDs, 2 ORs, and 4 chips.

Optimization 2: 'Double-Dipping' with Logic Gates

However, now that we have multiple ways each input can be written, this opens up more possibilities for one input to be written in terms of another.

We can compare all of these expressions to each other and look for similarities.

J1 = !q4*c1*!c2	
J2 = q1*!q4	
J3 = c2*!q1*!q2 + q4	J3 = q4 + c2*!c1*!q1
!K3 = !c2*q4	!K3 = c1*q4
J4 = c1*!c2*!q3 + c2*!c1*!q1	J4 = c1*!c2*!q3 + c2*!q1*!q2

Here, we found 2 duplicates.

If we compare the new K-map expression for J4 to the old K-map expression for J3:

J4 = c1*!c2*!q3 + c2*!q1*!q2 -> 1 OR, 4 ANDs

J3 = c2*!q1*!q2 + q4

We notice that they both have an identical c2*!q1*!q2 term. This means that we do not need to construct this gate logic twice. Once we already have it built for J3 in the circuit, we can simply wire that to the second input of the OR gate for J4. **This results in a savings of 2 ANDs.** It can be seen implemented in the Multisim section later. Note that this is a similar observation and would have led to the same result.

Through this process, we also notice something else.

J4 = c1*!c2*!q3 + c2*!q1*!q2

J1 = !q4*c1*!c2

Both the expressions for J4 and J1 have an c1*!c2 term. Following the same thinking above, we only need to construct that gate logic once. **This gives us a savings of 1 AND.**

In total, this optimization has saved us 3 ANDs – which brings our total down to 8 ANDs & 2 ORs, and actually saves us 1 AND chip!

<u>Using Previously Simplified Outputs to Decrease Number of Gates:</u>

Is there any further optimization we could do?

It can be noted here (and through the K-maps) that the simplified SOP expressions for J2 & !KC2 are identical. In that case, we can **simply wire !KC2 to J2**, and **save ourselves 1 AND gate that would have been used to construct !KC2.**

!KC2 = J2

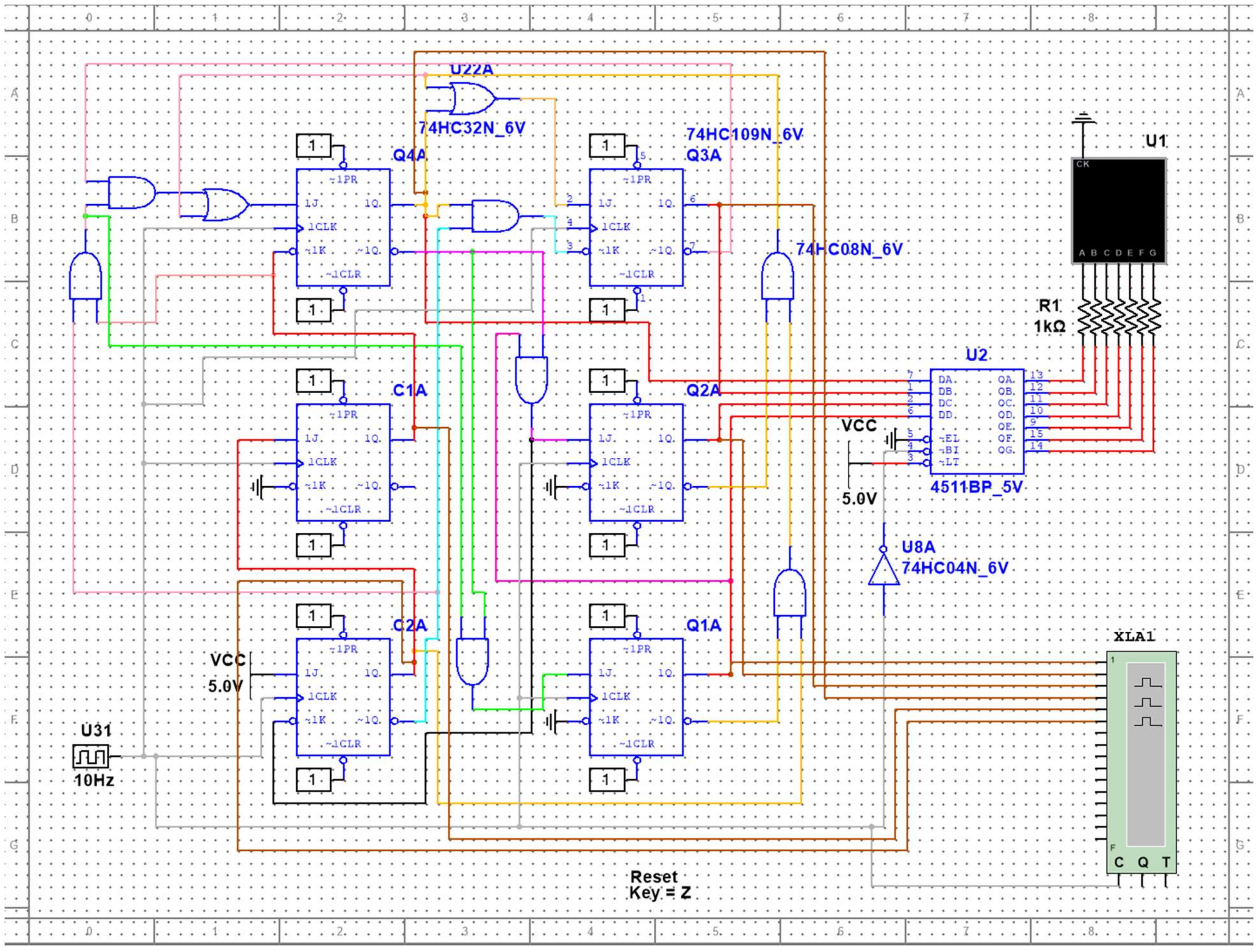
Final Summary:

Combining all 3 optimizations has resulted in a savings of 4 ANDs. The final implementation requires 7 ANDs, 2 ORs, and 3 chips.

These expressions were chosen to be used for the final design.

Input	Boolean Expression
J1	!q4*c1*!c2
!K1	0
J2	q1*!q4
!K2	0
J3	c2*!q1*!q2 + q4
!K3	!c2*q4
J4	c1*!c2*!q3 + c2*!q1*!q2
!K4	c1
JC1	c2
!KC1	0
JC2	1
!KC2	J2

MULTISIM:



Green: Gate input to J1

Hot pink: Gate input to J2

Mustard yellow: Gate input to J3

Salmon: Gate input to J4

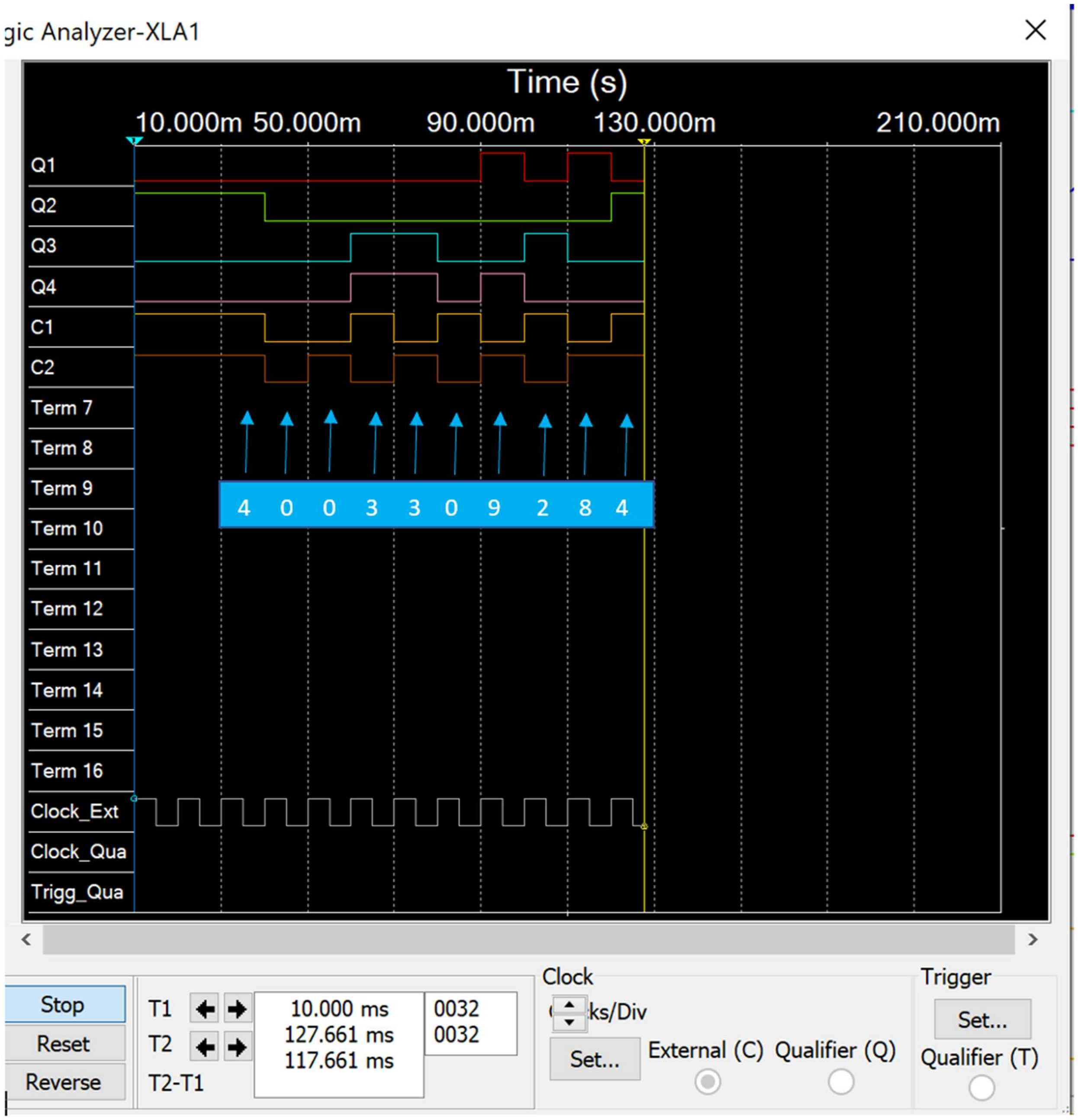
Black: Gate input to !KC2

Grey: Clock

Brown: Logic analyzer wires

Red: Other wires (simple inputs like !K4, connecting wires)

Note the interactive digital constants (on the preset and clear pins) are there to implement the reset feature of the FSM. In other words, to start the FSM at the first state 010011 (4).



T1	10.000 ms	0032
T2	127.661 ms	0032
T2-T1	117.661 ms	

This is the **timing diagram** generated on Multisim using the logic analyzer. It has been annotated here with the digits of my student number (which it loops through). It displays the bit values for Q1, Q2, Q3, Q4, C1, and C2 respectively which are compared below:

<u>Multisim</u>

4: 010011

0: 000000

0: 000001

3: 001110

3: 001101

0: 000010

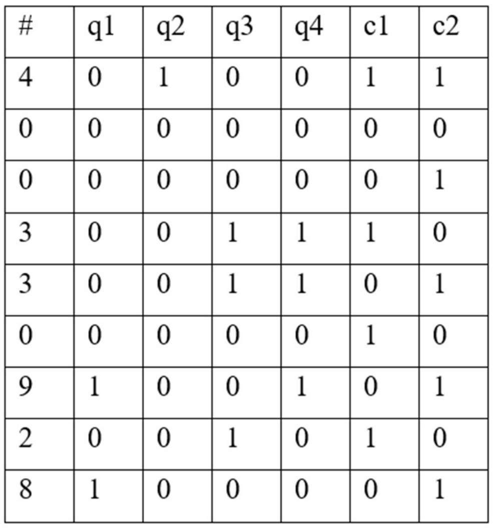
9: 100101

2: 001010

8: 100001

4: 010011

<u>Analytical:</u>



#	q1	q2	q3	q4	c1	c2
4	0	1	0	0	1	1
0	0	0	0	0	0	0
0	0	0	0	0	0	1
3	0	0	1	1	1	0
3	0	0	1	1	0	1
0	0	0	0	0	1	0
9	1	0	0	1	0	1
2	0	0	1	0	1	0
8	1	0	0	0	0	1

This shows that the timing diagram matches the values from the analytical section.

Please find a video of the Multisim simulation uploaded in the same Github repository.

PHYSICAL:

This part of the project was done in lab groups due to equipment constraints. Each group chose one student's design and physically implemented it using a breadboard and benchtop lab equipment. As such, this demo actually shows my friend's student number.

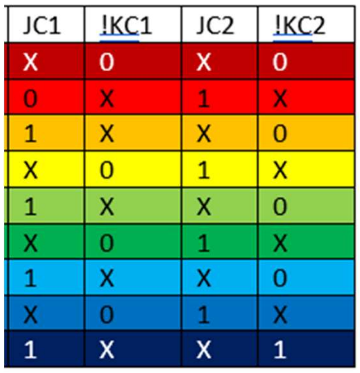
There were many troubleshooting and debugging difficulties we faced in building the circuit that took us several hours to solve. We used multiple strategies to do this, including testing different outputs with LEDs, going through the logic many times carefully, **examining timing diagrams to see which flip-flop was malfunctioning and comparing the actual & planned transitions to figure out the issue.** The root cause of the problem was discovered at the end to be a loose wire :D

Please find a video of the working physical implementation uploaded in the same Github repository, *with some commentary included.*

NEXT STEPS:

Some other optimizations I would have liked to explore, but could not due to time constraints include…

- Possibly reducing the number of gates by using NANDs in place of ANDs, ORs, and NOTs
- Exploring product-of-sums (POS) implementations
- Changing some of the counter values strategically to also reduce gates, for example:



JC1	!KC1	JC2	!KC2
X	0	X	0
0	X	1	X
1	X	X	0
X	0	1	X
1	X	X	0
X	0	1	X
1	X	X	0
X	0	1	X
1	X	X	1

(As in specifically assigning 0 or 1 to some of the Xs in these columns)