Nora Shao_L5A

October 26, 2023

1 Experiment 5 - Servo Motor Prep: Digital Basics

Name: Shao, Nora

Teammates at Table:

Student Number: [redacted]

Lab Section: L5A

[8]:
```
# @title
%%capture
from IPython.display import Image
from google.colab import drive
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

drive.mount('/content/drive/')
path = 'drive/My Drive/UBC ENPH Y2/ENPH259/Lab5/'
file_name = 'Nora Shao_L5A.ipynb'

assert file_name in os.listdir(path)

def img_path(img_name):
  return path + img_name
```

1.1 Prelab for Week 6: Truth Table for D-Latch

Problem Description: Given the D-latch in Figure 1a, construct a truth table.

Solution: I tried different inputs for D and E and different outputs for Q and E and checked for self-consistency of the logic. Figure 1b shows the resulting truth table I constructed.

[9]:
```
Image(filename = img_path("L5A_pre-lab d-latch.png"))
```

[9]:

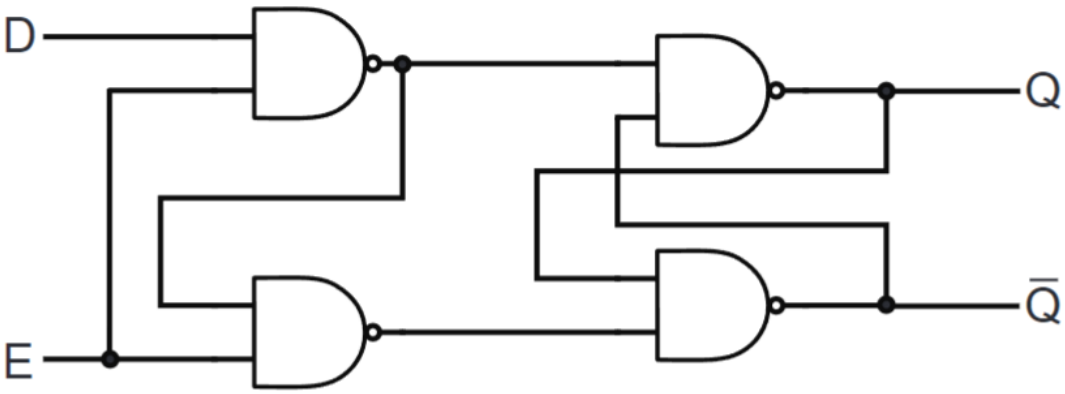
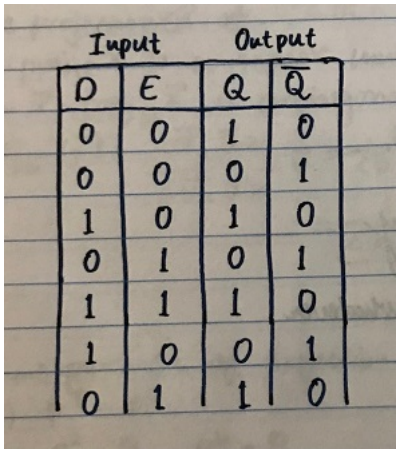


Figure 1a: A D-latch using *NAND* gates

[10]:
```
Image(filename = img_path("L5A_Pre-lab.jpg"))
```

[10]:



Input		Output	
D	E	Q	$\overline{Q}$
0	0	1	0
0	0	0	1
1	0	1	0
0	1	0	1
1	1	1	0
1	0	0	1
0	1	1	0

Figure 1b: Truth table for D-latch in Figure 1

2 Pre-lab for Week 7: R-2R ladder as a DAC

Problem Description: Describe how the equation

$$V_o = \frac{V_{b0}}{16} + \frac{V_{b1}}{8} + \frac{V_{b2}}{4} + \frac{V_{b3}}{2}$$

acts like a DAC converting the binary representation of bits 0-3 ($V_{b0} - V_{b3}$) to analog voltage V_0.

Solution: The circuit in Figure 2 acts as a basic DAC converter converting the digital values of 4 input bits to an analog V_o voltage. It follows the basic principle of mapping a binary number to its decimal equivalent, where each following 'bit' of a binary representation maps to a value twice

as large as the previous 'bit' when on. For example, say we have the 4-bit binary value 1101. This maps to the decimal value 2^3 + 2^2 + 2^0.

Accordingly, each input voltage bit on the DAC is divided by increasing factors of two, until they sum to an output analog voltage mapping to a decimal value. This way digital input voltages that can only be 0 or 1, or on or off, can map to an analog voltage.

[11]:
```
Image(filename = img_path("L5A_R-2R ladder DAC.png"))
```

[11]:

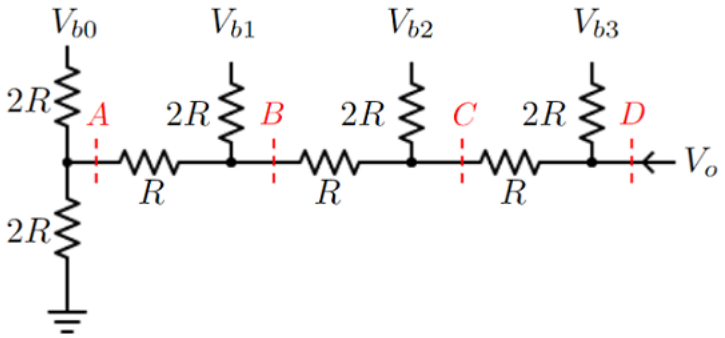


Figure 2: Resistive Ladder serving as a DAC. $V_{b0} - V_{b3}$ are the input bits and Vo is the analog output voltage. Nodes A-D are used for Thevinin equivalent analysis

3 Experiment

3.1 Schmidt-Trigger Inverter

Procedure: - First, without connecting anything to the breadboard, I attached the inverter to the breadboard across the gap bisecting the breadboard (so I'd have more space to connect wires in the same column as the inverter's terminals)

- Again, without connecting power, I pulled up the inverter's datasheet (https://www.mouser.com/datasheet/2/308/74HC14.REV1-34947.pdf) to set up the wires connecting its V_{cc} and GND terminals to the power and ground rails of the breadboard.
- For cleanliness, I first had two leads connected from the AD2 breakout board's V_{in} and GND terminals to the power and ground rails of the breadboard. The rails were then connected via wires to the V_{CC} and GND pins of the inverter to turn it on
- For the triangle wave input, I configured the WaveGen 1 output to a triangle wave with offset and amplitude 2.5 V
  - I connected the output via a wire from the probe of the output to the A1 terminal of the inverter
- I read the output from the inverter by connecting a BNC cable connected to the Scope 1+ input on the breakout board to a wire I then connected to the inverter's Y1 terminal
- Both of these were plotted together on WaveForm's scope

- I used the Y cursors to determine the input voltage that triggered the inverter's transtition from 0 to 1 states (visible in Figure [])
- I used WaveForms XY function to plot the input against output voltage, seen in Figure 7
- Then I found out that turning on the persistence setting of the XY graph gave me more datapoints and a more prominent graph

Troubleshooting

TS: There's an odd bump in my XY graph, but I was told to just ignore it because it was probably the inverter acting up.

[12]:
```
Image(filename = img_path("L5A_inverter io.png"))
```

[12]:

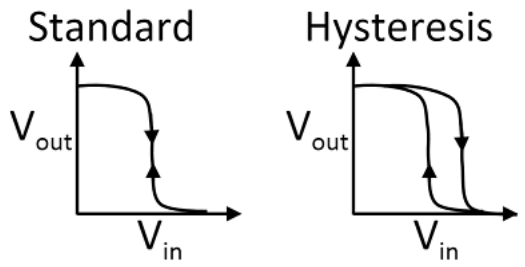


Figure 3: The input vs. output voltage of an inverter without and with hysteresis.

[13]:
```
Image(filename = img_path("L5A_inverter circuit.jpg"))
```

[13]:

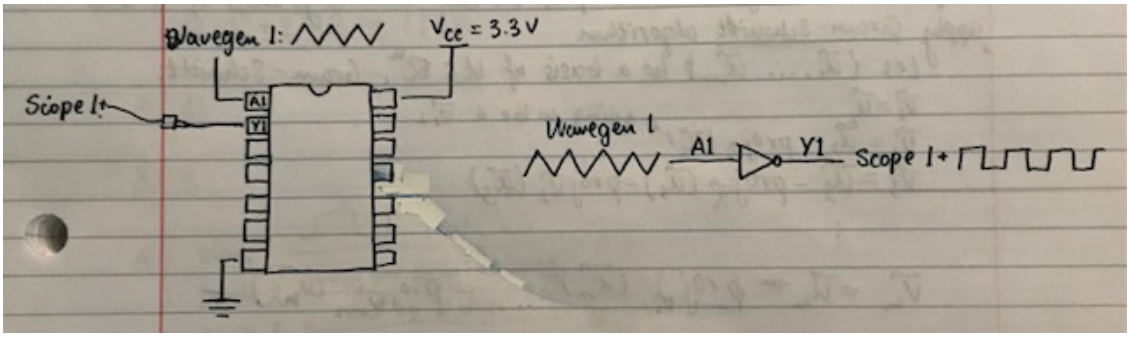


Figure 4: Circuit diagram of confirming operation of Schmitt-Trigger inverter on 74HC14A chip

[14]:
```
Image(filename = img_path("L5A_inverter setup.jpg"))
```

[14]:

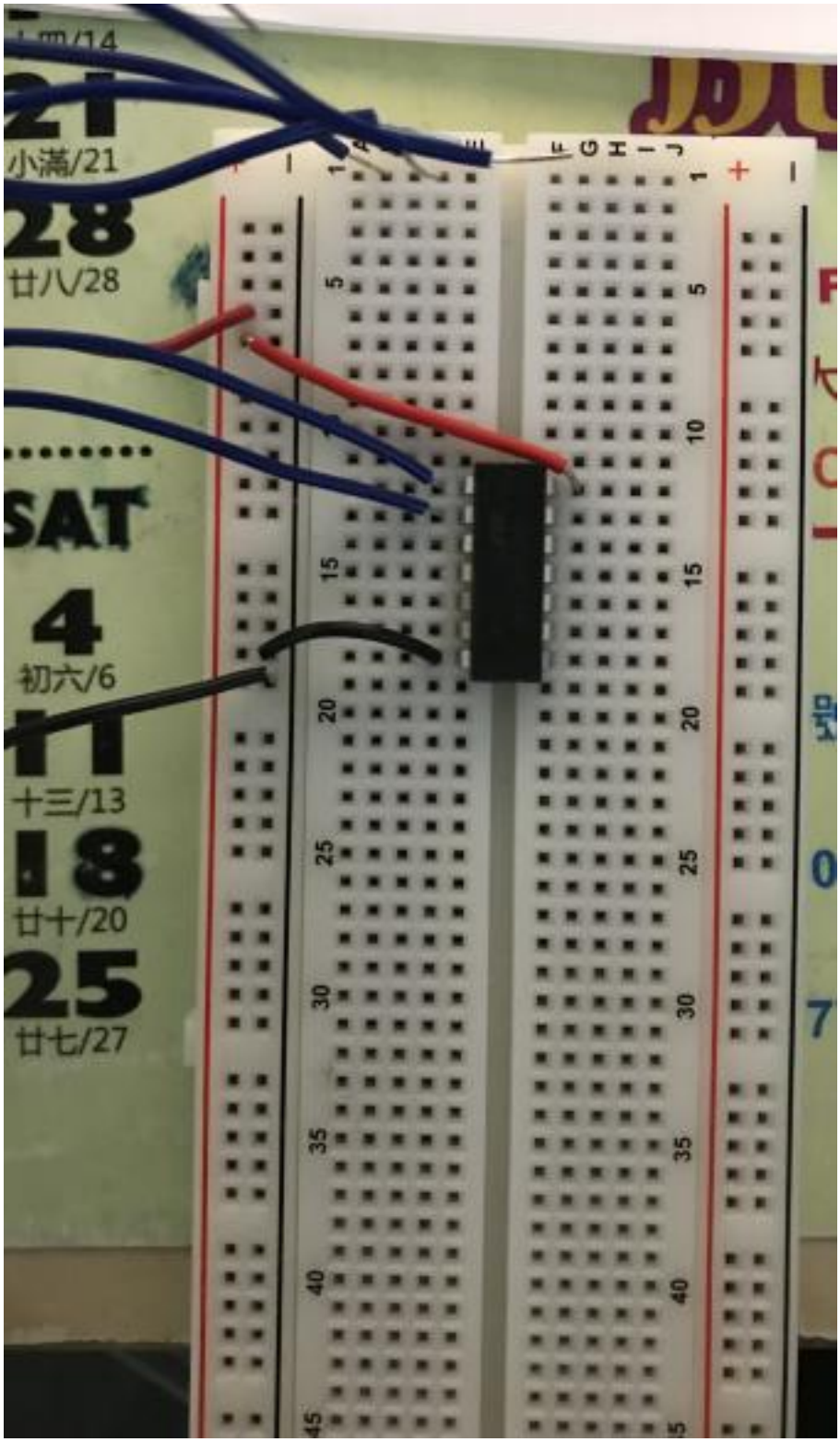


Figure 5: Setup of inverter on breadboard

Results: In Figure 6, we can see the continuous shifting of the inverter from the 0 to 1 (off and on) states as the input triangle voltage rises and falls.

Via the cursors in Figure 6, we see that the input voltage that triggers the inverter transitioning from 0 to 1 and the voltage that triggers the transition from 1 to 0 are different. On the rising edge of the input triangle wave, we see the signal from the inverter turn off (go to the 0 state) when the input reaches $1.8 \pm \frac{0.1}{\sqrt{6}}$ V. On the falling edge of the input triangle wave, we see the signal from the inverter turn on (go to the 1 state) when the input falls to $1.1 \pm \frac{0.1}{\sqrt{6}}$ V.

When I set up the X-Y graph (visible in Figure 7) while the input signal was operating at 10 kHz, the rising and falling edge of the

When I lowered the input's frequency to ~100 Hz, the on and off in the X-Y graph straightened significantly, but there were less datapoints, hence why I enabled the persistence function to get a clearer graph.

Understanding:

In Figures 6 and 7, we observe how the input voltage that turns the inverter output on and the voltage that turns the inverter off are different, and how in the XY graph there are two separate paths for the inverter's output (or inversion) depending on whether the input voltage is above or below the threshold of what the inverter considers the 1 (on) state.

This occurs because the Schmitt-Trigger inverter we are using uses hysteresis to mitigate the noise from the input voltage. Figure 3 explains what we see in the XY graph in Figure 7 better, showing the behaviour of the inverter as the input voltage changes, with and without hysteresis. What we see in Figure 7 in the graph on the right, with the inverter taking separate 'paths' inverting 'on' (above ~1.8 V) voltages to off and inverting 'off' (below 1.1 V) voltages to the on state, as, well, 1.8 V and 1.1 V are not equal. The hysteresis sets a threshold the input voltage must meet/ be in-between to flip. This way any input voltage, which consists of a range of values, that meet these requirements will flip the inverter, improving its accuracy, much like how digital signals work. The inverter doesn't rely on an exact input voltage to operate, and the different on- and off-voltages the inverter requires to flip from HI to LO or vice versa also prevents it from being confused, kind of like a debounced pushbutton.

[15]:
```
Image(filename = img_path("L5A_task 1 corrector measurements.png"))
```

[15]:

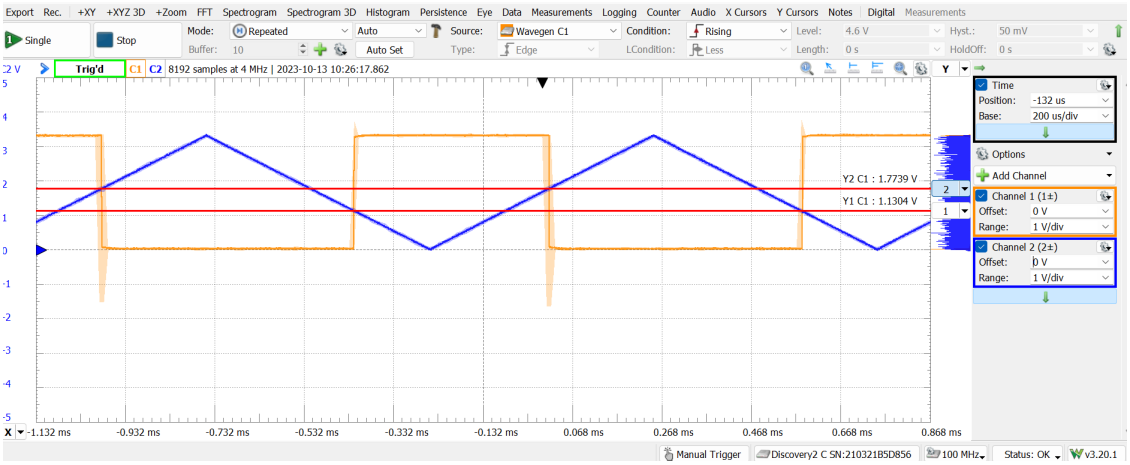


Figure 6: Operation of inverter; chip inverts input (blue) to on-off states (orange) once input crosses threshold.

[16]:
```
Image(filename = img_path("L5A_task 1 xy persistence (annotated).jpg"))
```

[16]:

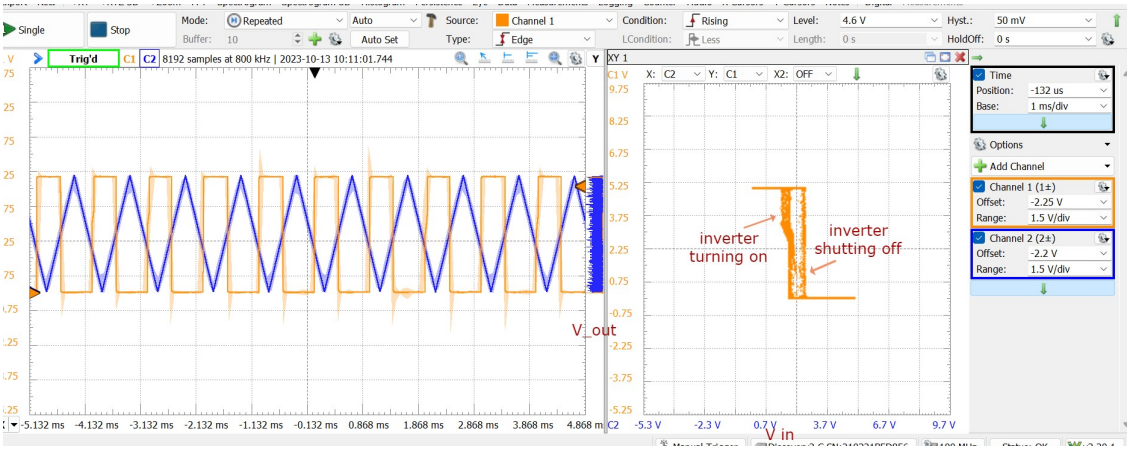


Figure 7: Input (x-axis) vs. output (y-axis) voltage of inverter plotted against each other

3.2 D-latch

Construct a D-latch from NAND gates in a 74HC00 chip.

Procedure:

Checking Operation of NAND: Confirm the truth table of a NAND using one gate in the 74HC00 chip

- I disconnected everything from the breadboard, then plugged the NAND gate (chip) in across the gap bisecting the breadboard

- I followed this datasheet (https://www.mouser.com/datasheet/2/308/74HC00-105628.pdf) to figure out where to place my input and outputs
- I set up the Digital I/O terminals for the two inputs and one for the output of the NAND gate. This setup is visible in Figure 9, and an explanation in Figure 10.
  - I have DIO 0 and 1 connected to the NAND chip's A1 and B1 terminals, and DIO 2 to the output terminal Y1.
- Now that we have DIO 0 and 1 connected to the input of the NAND gate and DIO 2 reading the output, I configure them in Waveforms.
  - This configuation in visible in Figure 12. I set up DIO 0 and 1 to pulse at different frequencies so I could get all 4 possible input states (since each of the two inputs can take two values). In my case, I had DIO 0 pulsing at 500 Hz and DIO 1 at 1 kHz.

Actual D-latch

- After disconnecting the wires from the previous part (checking the operation of a NAND gate), I had to figure out how to connect the terminals of the four NAND gates on the chip to set up a D-latch.
- Using the diagram in Figure 1a, I setup the diagram on the left of Figure 10 to help direct myself on which inputs to wire to which outputs and vice versa. It was quite confusing.
- The finished wiring job can be seen in Figure 11.

Note: From now on, I will refer to D in Figure 1a as Data and E as Enable. Q is still Q, but Q' represents Q complement. Also, the states HI/1/on are the same, and so are LO/0/off.

Troubleshooting:

TS: I couldn't figure out why my output in the Logic window was a constant 0 V when my output channels were evidently turning on and off (which I controlled via 'Supplies' on WaveForm). I disconnected my power supply from the breadboard and the LED turned on. Turns out (with the help of the teacher), I had connected my ground supply to the power rail instead of the ground rail and basically shorted my power straight to ground.

TS: I also found out I had choose the frequencies I pulsed my Data and Enable in digital I/O pins wiser.

First, I shouldn't pulsing my Data and Enable inputs not at frequencies that are not a multiple of 2. I first had Enable's frequency as half of Data's, which made me miss out on two of the possible states of the d-latch. The d-latch has 6 possible states between Data, Enable, Q, and Q complement, but (all visible in Figure 14), I was only seeing 4 because of the perfect overlap between Enable and Data's frequency.

Second, for clearer information from the data, I needed to set Enable's frequency to something (significantly) lower than Data's. I originally had the the input Data and Enable and outputs Q and Q bar depicted in Figure 13, where Enable was turning on and off at a frequency twice that of Data's. This is bad because Enable pulsing faster than data means you can't actually see how Q behaves at different states of Enable; Enable 'controls' Q via Data, and it's clearer if there are longer periods of Enable being on and off as Data and Q change. Figure 14 shows what we want better.

With Mr. Jones's help, I set Data to pulse at a frequency of 3300 Hz and Enable at 400 Hz. As seen in Figure 14, this made the different states of Q (and Q') depending on Data and Enable much clearer.

[17]:
```
Image(filename = img_path("L5A_NAND circuit.jpg"))
```

[17]:

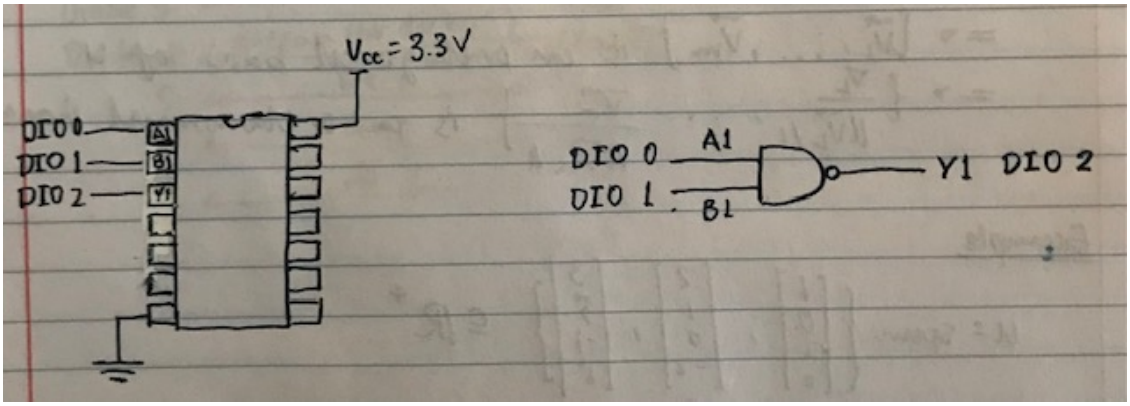


Figure 8: Circuit diagram of confirming operation of NAND gate on 74HC00 chip

[18]:
```
Image(filename = img_path("L5A_NAND setup.jpg"))
```

[18]:

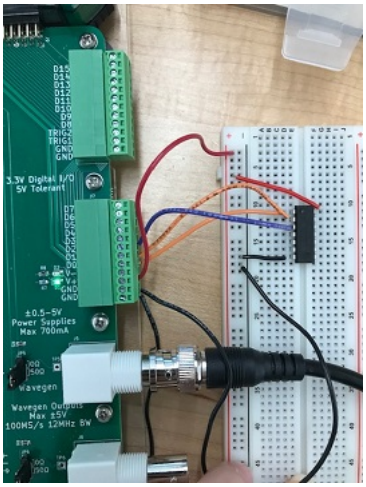


Figure 9: Breadboard setup of NAND gate

[19]:
```
Image(filename = img_path("L5A_d-latch circuit.jpg"))
```

[19]:

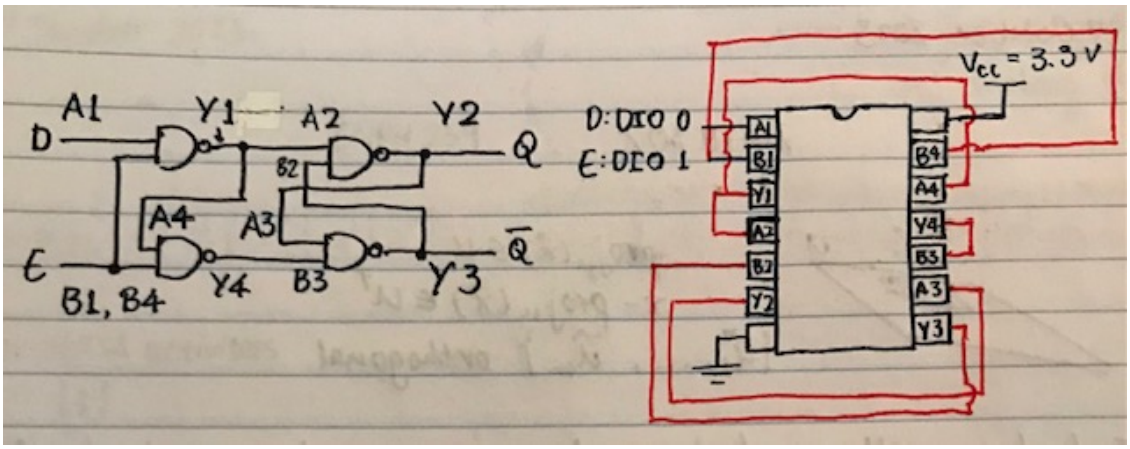


Figure 10: Translating d-latch in Figure 1a to four NAND gates on 74HC00 chip

[20]:
```
Image(filename = img_path("L5A_d-latch setup.jpg"))
```

[20]:

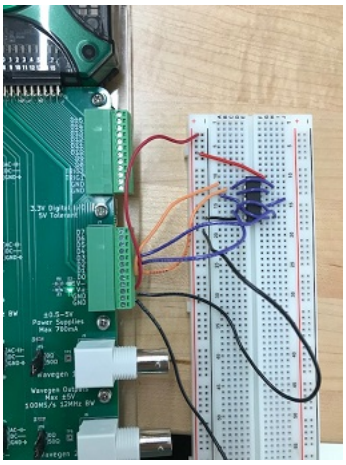


Figure 11: D-latch setup on breadboard

Results: As seen in Figure 14, we have four channels in the Logic window for Data, Enable, and Q and Q'. We observe that when Enable in on/high, Q reflects whatever state Data is at. When Enable turns off, Q stays at whatever state Data was at right before Enable (so on the last little edge of Enable being high). As well, we see that Q' looks like Q mirrored - always opposite of the state of Q, which makes sense, as the two should always be opposites.

The states show that the truth table I made in Figure 1b isn't entirely correct.

Instead, Figure 14 shows us the possible states of the d-latch as

1. Data: 1, Enable: 1, Q: 1, Q': 0
2. Data: 0, Enable: 1, Q: 0, Q': 1
3. Data: 0, Enable: 0, Q: 0, Q': 1
4. Data: 0, Enable: 0, Q: 1, Q': 0
5. Data: 1, Enable: 0, Q: 0, Q': 1
6. Data: 1, Enable: 0, Q: 1, Q': 0

The most important takeaways from this table is that whenever Enable is 1, Q is Data, and Q' is always the opposite of Q.

Understanding: The d-latch behaves as a door. When Enable is on, the d-latch is an open door; Q just reflects whatever state Data is at and you can 'see through' the door to the other side. When Enable turns off, the door/d-latch closes, and Q stays at the whatever state Data was at right before Enable turned on. Keeping to the analogy, you keep in your mind what you saw past the door before it closed.

This way, you can use the d-latch to store 1-bit information; keep Data at whatever state (on or off, 1 or 0) you want to record in Q, have Enable start on and then turn it off, and Q will store the value in Data indefinitely.

In my truth table for the d-latch in Figure 1b, I had an extra state at the bottom that wasn't possible, since Q must always reflect Data whenever Enable is HI/1/on.

[21]:
```
Image(filename = img_path("L5A_working NAND.png"))
```

[21]:

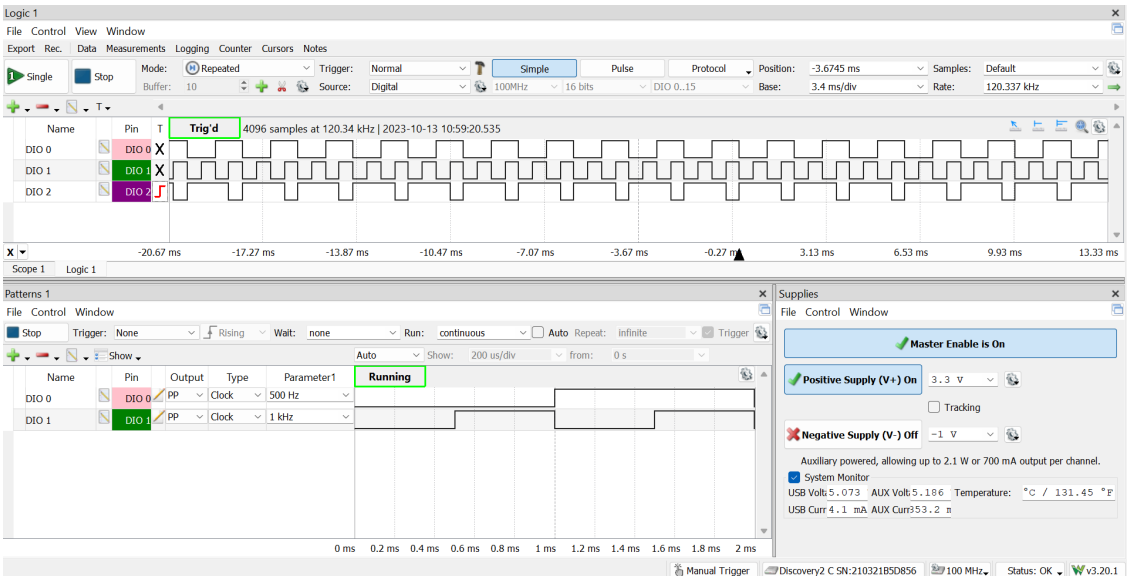


Figure 12: Confirmation of working NAND gate with DIO 0 and 1 as inputs and DIO 2 as the NAND's output.

[22]:
```
Image(filename = img_path("L5A_renamed working d-latch.png"))
```

[22]:

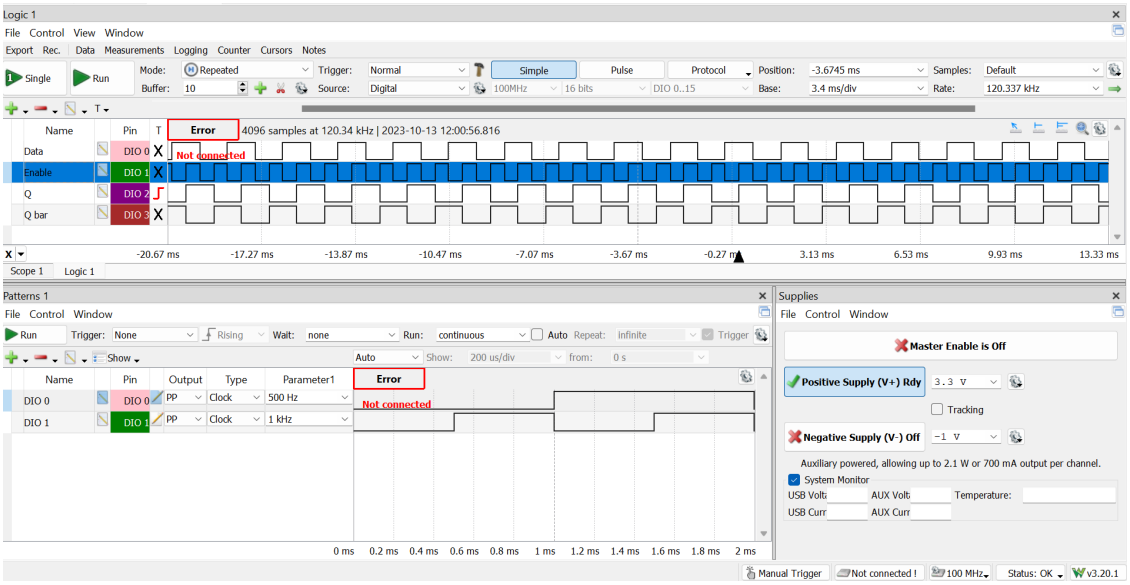


Figure 13: Somewhat erroneous input Data and Enable (DIO 0 and 1) signals and ouputs (Q and Q') from the d-latch

[23]:
```
Image(filename = img_path("L5A_actual working d-latch with q.png"))
```

[23]:

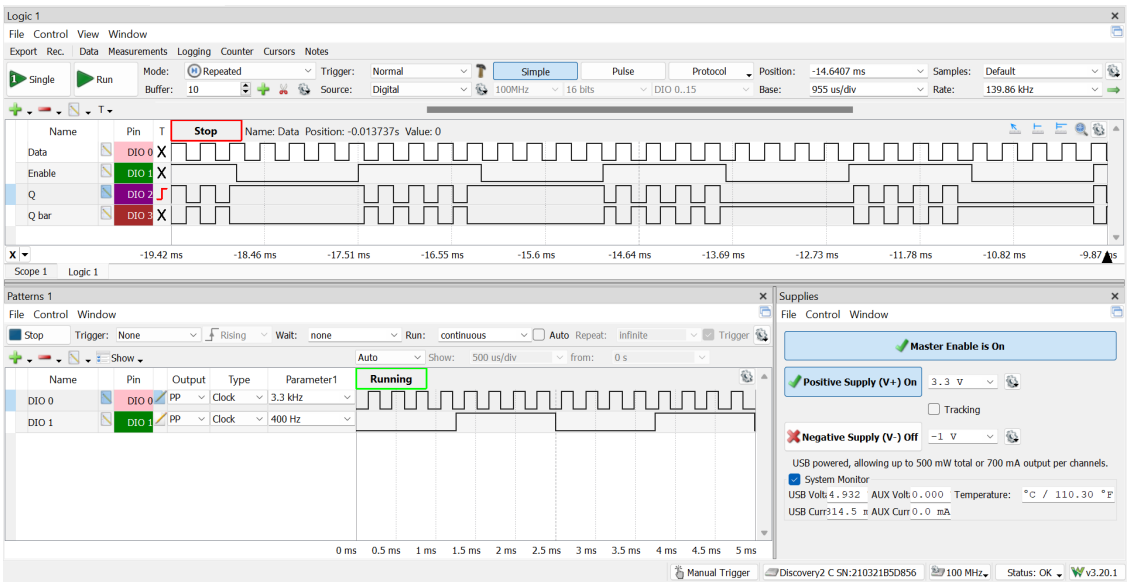


Figure 14: Correct input (Data as DIO 0 and Enable as DIO 1) and outputs from d-latch, showing 6 possible states

3.3 DAC

Procedure: - I pull up the datasheet for the provided resistive ladder here: https://www.bourns.com/docs/Product-Datasheets/r2r.pdf

- I set up the breadboard as seen in Figure 16 following the circuit diagram in Figure 15.
  - I designed the setup following Figure 2 as a guide

[24]:
```
Image(filename = img_path("L5A_sawtooth circuit.jpg"))
```

[24]:

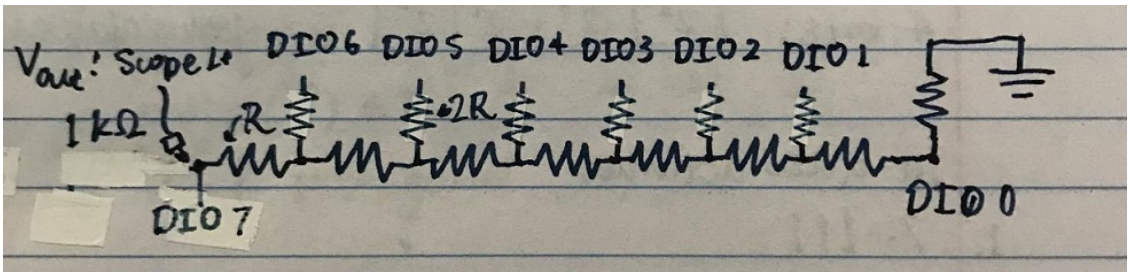


Figure 15: Schematic of input and output pins connected to resistor ladder for sawtooth wave

[25]:
```
Image(filename = img_path("L5A_sawtooth wave setup.jpg"))
```

[25]:

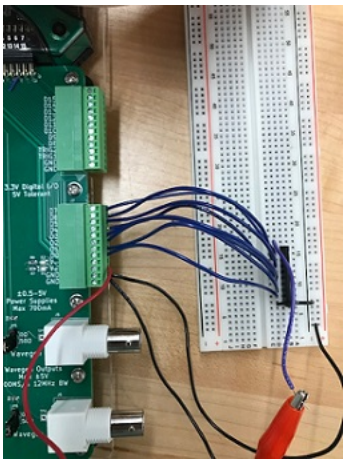


Figure 16: Setup of resistor ladder on breadboard for sawtooth wave

[26]:
```
Image(filename = img_path("L5A_sawtooth pattern.png"))
```

[26]:

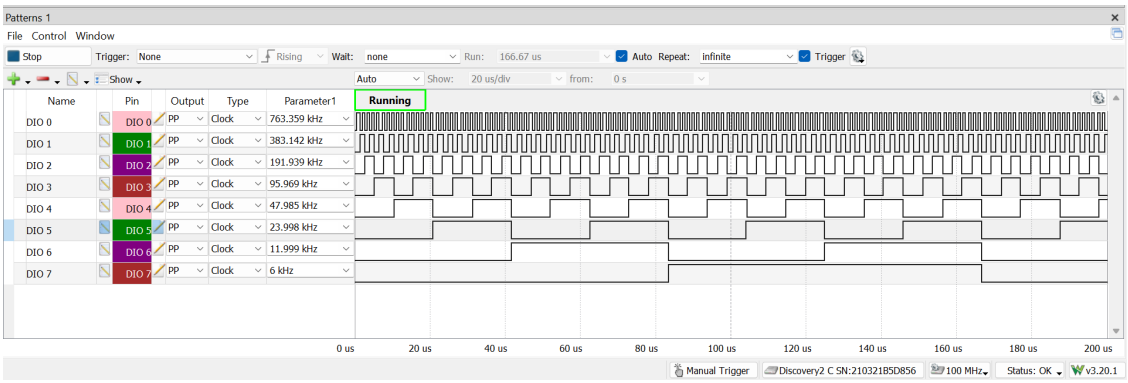


Figure 17: Signals sent into input pins connected to resitive ladder

[27]:
```
Image(filename = img_path("L5A_sawtooth with load circuit.jpg"))
```

[27]:

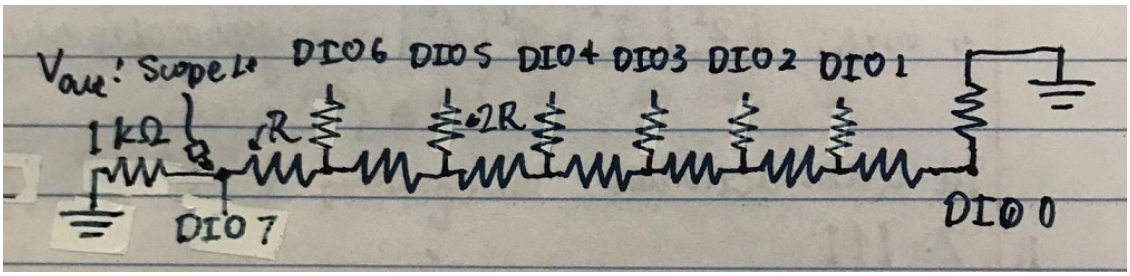


Figure 18: Schematic of input and output pins and load resistor (1 $k\Omega$) connected to resistor ladder for sawtooth wave

[28]:
```
Image(filename = img_path("L5A_sawtooth wave and load setup.jpg"))
```

[28]:

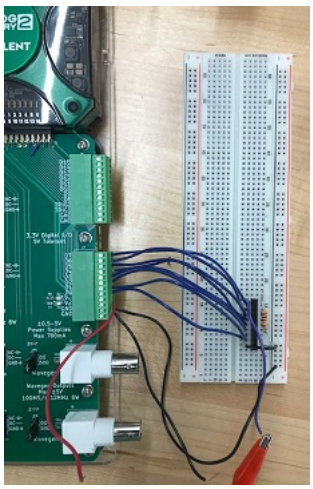


Figure 19: Setup of resistor ladder on breadboard and load resistor for sawtooth wave

Results:

Indeed, in Figure 20, we read a sawtooth wave from V_{out}.

When we add the resistor load of 1 $k\Omega$ between V_{out} and ground, I immediately saw a significant decrease in signal strength/voltage from V_{out}. We see this in Figures 21 and 22. However, V_{out} kept its sawtooth shape.

Understanding:

To manipulate my input bits to have the resulting voltage increase linearly, I had to understand how binary counting worked. As well, I relied on my understanding that the resistive ladder was a linear circuit. I figured out that when binary numbers increase linearly, each following bit takes twice the time to increase (from 0 to 1) than the preceding bit; for example the second digit of a binary number takes 2^1 counts to go from 0 to 1, the third takes 2^2 counts to go from 0 to 1, the fourth takes 2^3, etc. So I just applied this knowledge to the digital input pins of the resistive ladder, incrementing/pulsing each succeeding digital input further from V_{out} at double the frequency. DIO 7, as seen in Figure 15 and 18, are closest to V_{out} and thus have the slowest frequency since they represent the highest digit of the binary number that increments the slowest. DIO 7 is the most significant digit because its voltage is the least 'diluted' by the resistive network. Through Thevenin analysis, I understood how the magnitude of each succeeding digital input was reduced by a factor of two due to the resistors. Since the circuit is linear and I wanted a frequency of 6k Hz for the sawtooth wave from V_{out}, I just pulsed DIO 7 at the base 6 kHz and multiplied each succeededing digital input pin by 2.

The reason V_{out} held its shape but decreased significantly in amplitude is because the load resistor between V_{out} and ground effectively acts as a voltage divider. So while originally all the digital input voltage through the pins were transformed via the resistive ladder to an analog V_{out}, now most of the voltage goes through the load resistor, and V_{out} only sees a fraction of the resistance. It's possible to calculate the exact resistance of the resistive ladder components since we have the amplitude of the new V_{out} and the load resistor.

The resolution of the DAC is the smallest value it can increment by, which would be the incrementation of its least signficant bit, DIO 0. This corrects to the value of 1 in terms of binary numbers. Since I don't have the resistive ladder with me anymore, I can't determine what exactly this value is, but it is determined by the value of DIO 0's HI voltage and the resistance of the resistors in the 2-2R ladder.

[29]:
```
Image(filename = img_path("L5A_sawtooth wave.png"))
```

[29]:

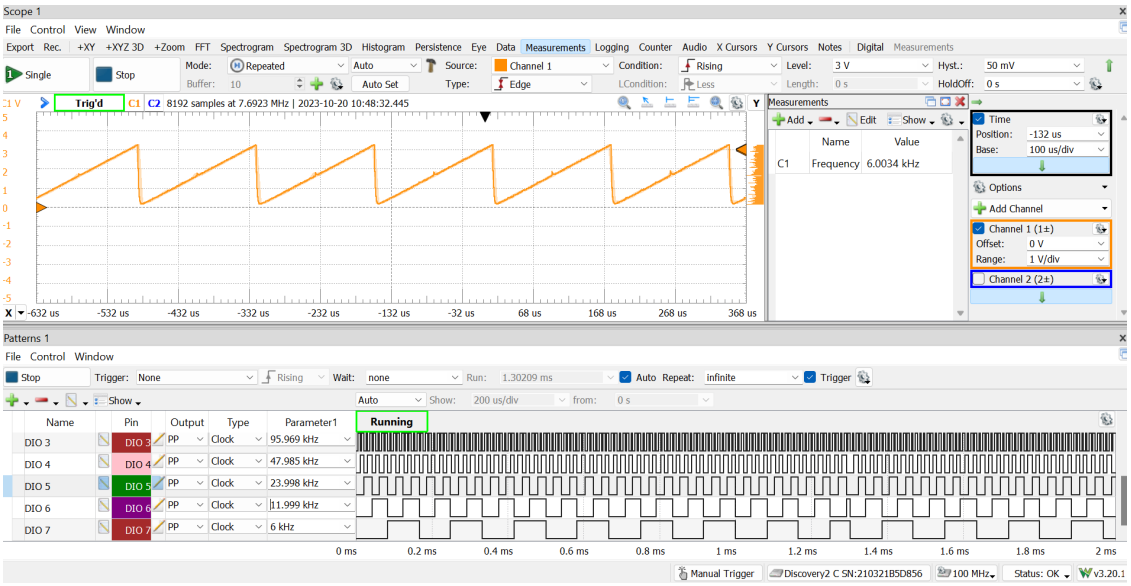


Figure 20: Sawtooth wave outputted from resistive ladder and digital input pins

[30]:
```
Image(filename = img_path("L5A_sawtooth and resistor.png"))
```

[30]:

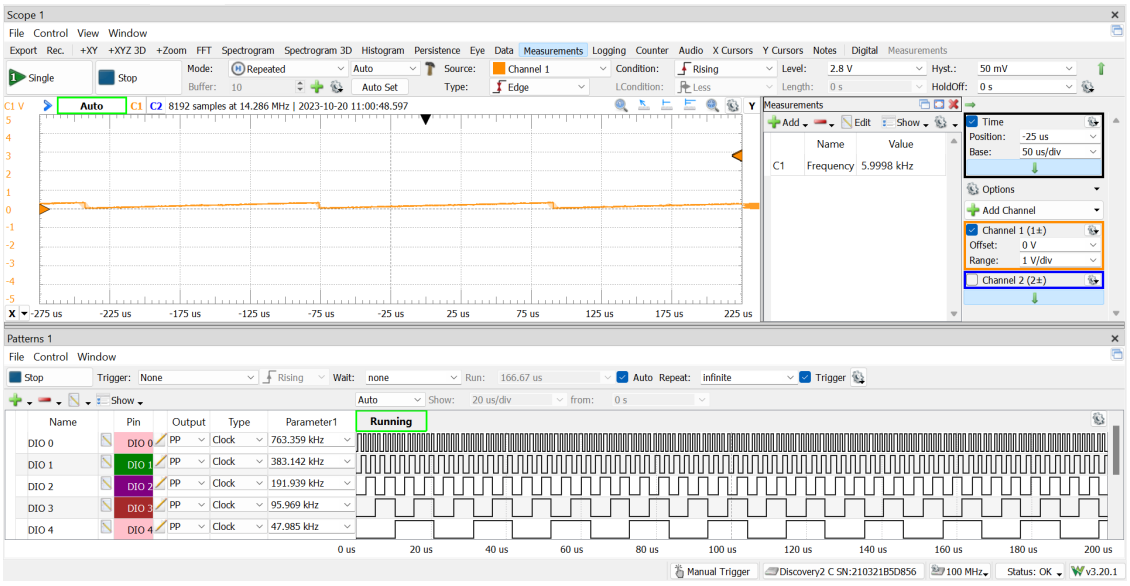


Figure 21: Sawtooth wave outputted from resistive ladder and digital input pins with addition of load resistor

[31]:
```
Image(filename = img_path("L5A_zoomed in sawtooth and resistor.png"))
```

[31]:

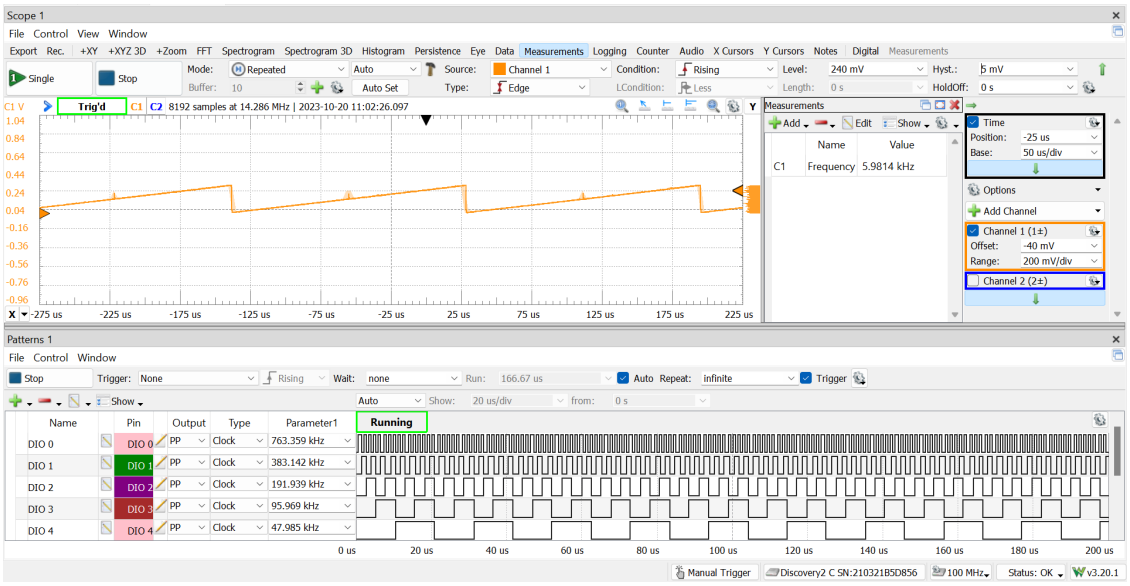


Figure 22: Sawtooth wave outputted from resistive ladder and digital input pins with addition of load resistor (zoomed in)

3.4 Counter

3.4.1 ÷ 2 Counter

Configure one of the ÷2 counter ICs to count using a series of negative pulses and a lower frequency (or even a manual switch) provided by the Patterns app)

Procedure:

- I use this datasheet for the 74HC390 counter: https://www.onsemi.com/pdf/datasheet/mc74hc390-d.pdf
- I first turned the chip on by adding a wire from the AD2 with 3.3 V to the chip's V_{CC} port, then I grounded the chip's GND terminal by connecting a grounding wire from the AD2's GND terminal to the grounding rail of the breadboard, and extending that ground with another wire to the chip's GND terminal.
- I then connected a 1kHz on-off clock to the Clock pin via a wire from DIO 0.
- I connected DIO 1 to the Reset terminal of the chip and then, in WaveForm's Patterns, just set that to perpetually off (since I don't really need to reset the ÷ counter; it'll reset itself once it goes past 0 and 1).
- I connected DIO 2 via a wire to the chip's Q_{Aa}; this reads the counter
- In WaveForm's Logic feature, I read DIO 2 and saw it turn on and off as it counted from 0 to 1 again and again.

Troubleshooting: Looking at Figure 25, and how the counter increments with the

[32]:
```
Image(filename = img_path("L5A_divide by two circuit.jpg"))
```
[32]:

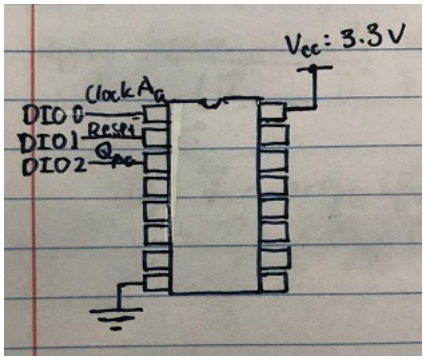


Figure 23: Circuit diagram for using the ÷ 2 section of a ripple counter

[33]:
```
Image(filename = img_path("L5A_dvidie by two setup.jpg"))
```
[33]:

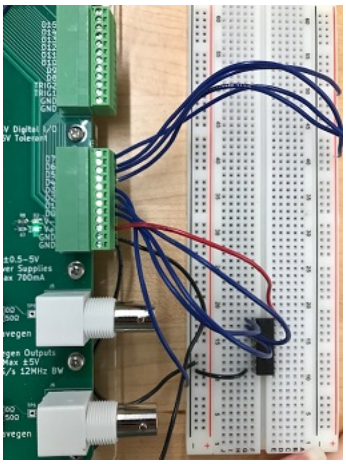


Figure 24: Breadboard setup of divide by two counter

Results & Understandings: As visible in Figure 25, we have a working divide by two counter. DIO 0, connected to the chip's Clock A_a pin, pulses steadily with every two pulses turning the state of the counter (DIO 2), which is connected to the Q_{Aa} pin, to a different state. More clearly, every two times the clock turns off or we see a falling edge, the counter increments 1, or turns on, hence why it's a ÷ 2 counter. Since this is how we expect it to work, from the outputs in Logic in Figure 25, we know our counter is working properly.

[34]:
```
Image(filename = img_path("L5A_divide by two.png"))
```
[34]:

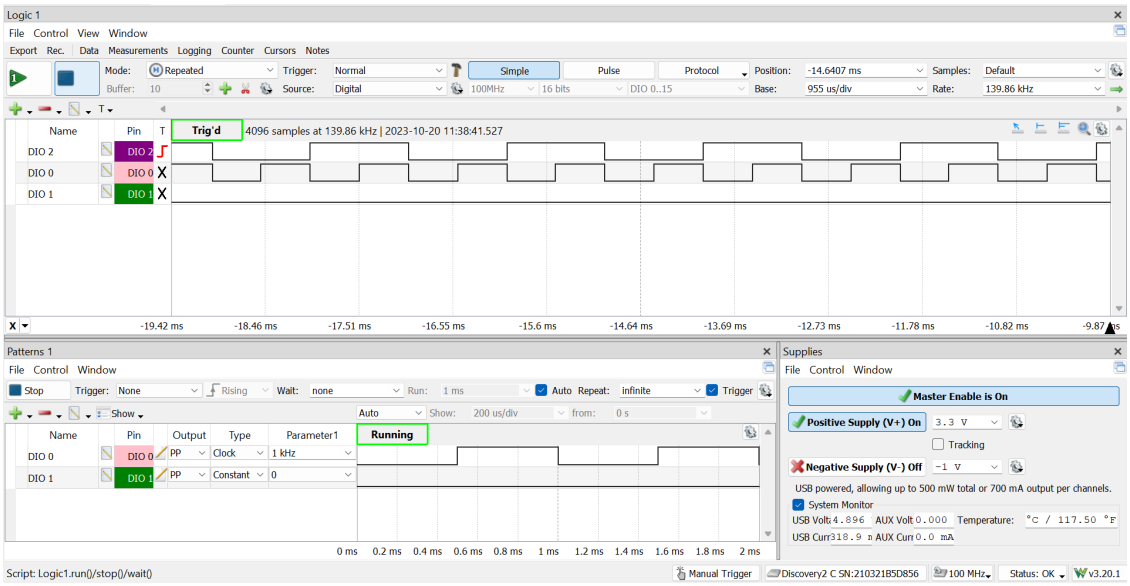


Figure 25: Working ÷ 2 counter, with DIO 0 as the input clock and DIO 2 as the output counter

3.4.2 ÷ 10 Counter

Make a divide by 10 counter using the ripple counter chip

Procedure

- I followed the same procedure as for the ÷ 2 counter, but added the ÷ 5 counter as well since $2 \times 5 = 10$.
- I set up DIO 0 as the Clock input to the divide by 5 clock (Ba) with a frequency of 1 kHz
- I connected each of the three bits for the ÷ 5 counter on Side A of the chip together, then I connected the most significant bit, the last one/Q_{Da} to Clock A_a, which controlled the divide by 2 counter
- Then I read the output from Q_{Aa} via DIO 2

Troubleshooting: Again, I found I had to reset the counter every 100 Hz (a tenth of the input frequency to work).

[35]:
```
Image(filename = img_path("L5A_divide by 10 circuit.jpg"))
```

[35]:

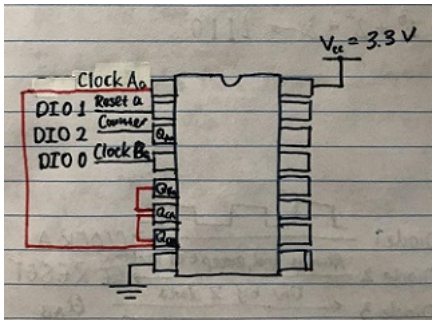


Figure 26: Circuit for divide by 10 counter on chip

[36]:
```
Image(filename = img_path("L5A_divide by ten setup.jpg"))
```

[36]:

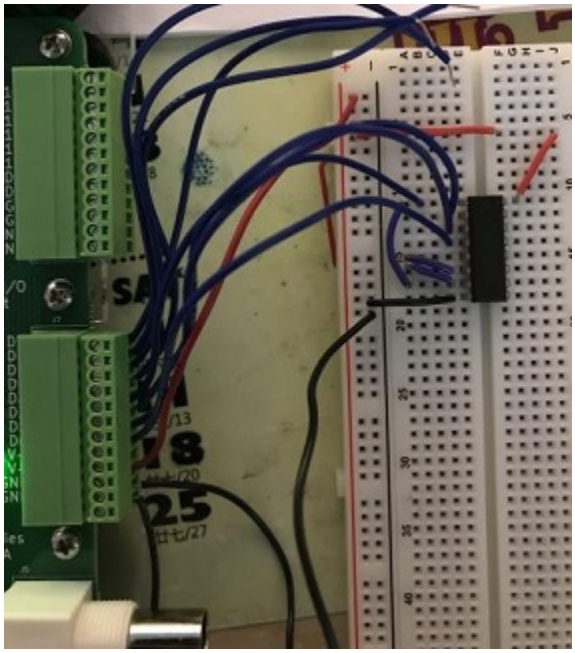


Figure 27: Breadboard setup of divide by 10 counter

Results & Understandings: Figure 28 depicts our divide-by-10 counter output. As can be seen, the counter on the Logic view increments once every time the clock turns off 10 times. This tells us our divide by 10 counter is working.

[37]:
```
Image(filename = img_path("L5A_divide by 10 issue.png"))
```

[37]:

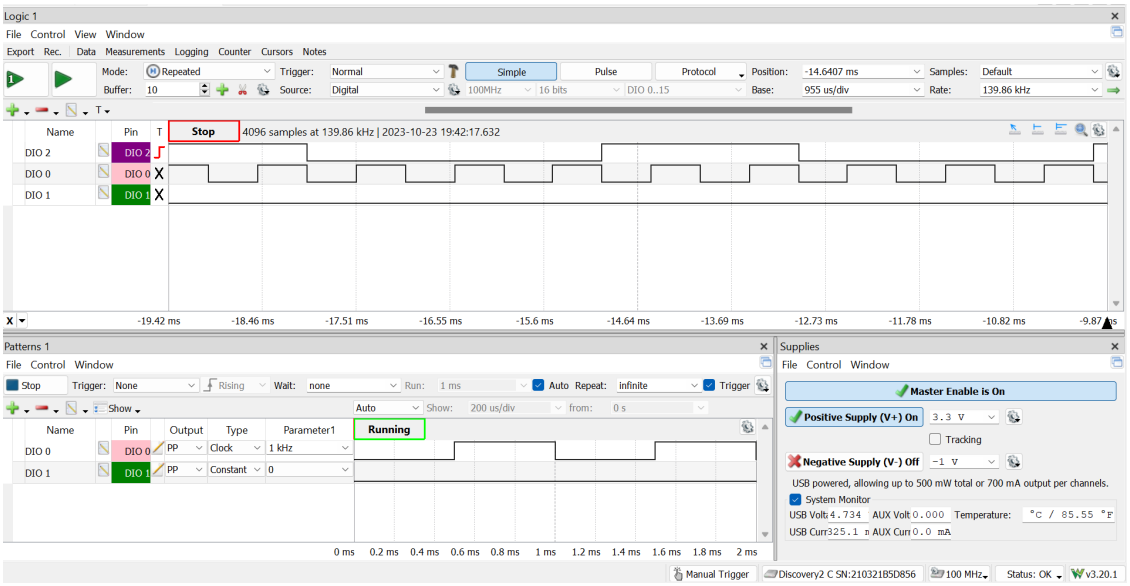


Figure 27: Incorrect divide by 10 output

[39]:
```
Image(filename = img_path("L5A_working divide by 10.png"))
```

[39]:

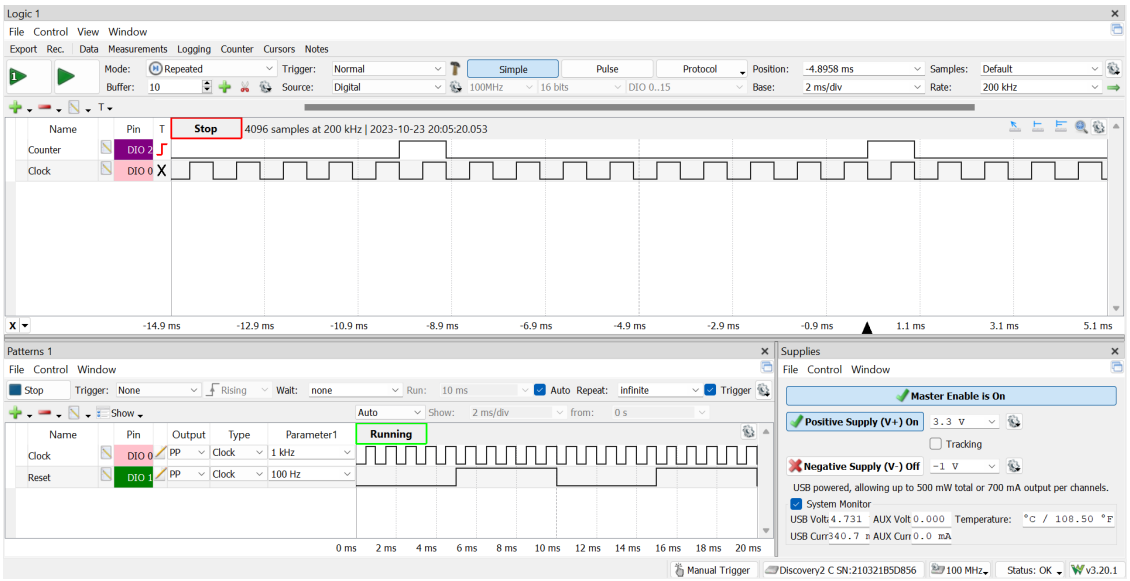


Figure 28: Working divide by 10 counter on WaveForms Logic

3.4.3 Divide by 100

Divide by 100 counter

Procedure

- I set up the chip on the breadboard across the gap.
- I powered it with the 3.3 V supply from the AD2 breakout board. I configured this on WaveForms, as seen in Figure 31's bottom right.
- I grounded the bottom left (ground) terminal of the chip.
- Each side of the chip is its own divide by 10 counter - as demonstrated above, so if I connect them to each other, I get a $10 \times 10 = 100$ counter.
- I connected the output of the left-hand side divide by 5 to the left-hand side divide by 2 input. I connected the divide by 2's output to the right-hand side's divide by 3 input clock, then connected that to the right-hand side's divide by 2 input clock. Then I read the right-hand side's divide by 2 output as the divide by 100 counter.
- I read the output with the breakout board's DIO 3
  - This is explained in the diagram in Figure 29.
  - The finished setup is visible in Figure 30
- I had DIO 0 connected to Clock Ba, which input into the left's divide by 5 counter, which I set to 1 kHz.
- I confirmed the operation of the counter by measuring the frequency of the counter on WaveForms relative to the frequency of the clock (DIO 0). Since it's a divide by 100 counter, the frequency of the output counter should be 1/100 of that of the input. Since the input is 1 kHz, the frequency of the counter should be 10 Hz.

Troubleshooting

TS: Initially, I measured a counter frequency of 20 Hz, which is problematic because it basically indicated I had a divide by 50 counter instead of divide-by-100. However, I figured maybe the issue was the same as for my divide by 10 counter, where I actually needed to reset some of the counters. Since the right side of the chip is its own divide-by-10 counter, the left hand side of the chip (Side A) would be at most operating at 1/10th the frequency of the right. So I figured I would reset it at something slightly less than 100 Hz, such as 50 Hz.

This actually did work, and I measured a counter frequency of 10 Hz, which is what I wanted, as 100×10 Hz $= 1$ kHz, which is what my input clock operated at.

[40]:
```
Image(filename = img_path("L5A_divide by 100 circuit.jpg"))
```

[40]:

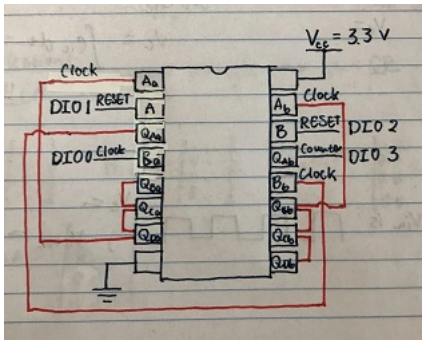


Figure 29: Divide by 100 on chip circuit connections

[41]:
```
Image(filename = img_path("L5A_divide by 100 setup.jpg"))
```

[41]:

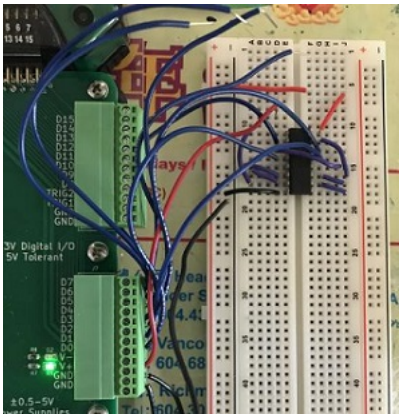


Figure 30: Divide by 100 setup on breadboard

Results & Understandings:

In Figure 32, we have a working divide-by-100 counter. I constructed it by breaking it down as a product (literally) of the previous counters I made. Since I know the divide by 10 counter is just the divide by 5 and divide by 2 counter multiplied/combined, I figured the divide-by-100 counter is just two divide-by-10-counters, so I would have to wire together all the individual counters in the ripple counter chip we have.

I planned to confirm the operation of counter by ensuring for every 100 pulses of the input clock, the counter would increment once. This is how I noticed the output in Figure 31 was incorrect; the counter had a frequency of 20 Hz, which meant it was incrementing every 50, not 100 pulses. I figured out this problem by implementing a reset for the divide-by-10 counter on Side A of the chip. At this point, I realize that maybe I wired my divide by 10 and divide by 100 counters incorrectly by connecting the three bits of the divide by 5 counter, but I don't have the necessary materials to fix that anymore.

This explains why I needed to reset my counters for them to work; initially I figured I wouldn't need to reset anything because the counters would reset themselves once they were past their maximum value. As well, this would explain why the divide by 5 counter needed three bits, as binary numbers neede three bits to store values that correspond to a decimal value of 5 (101) and above,

[42]:
```
Image(filename = img_path("L5A_divide by 100 issue.png"))
```

[42]:

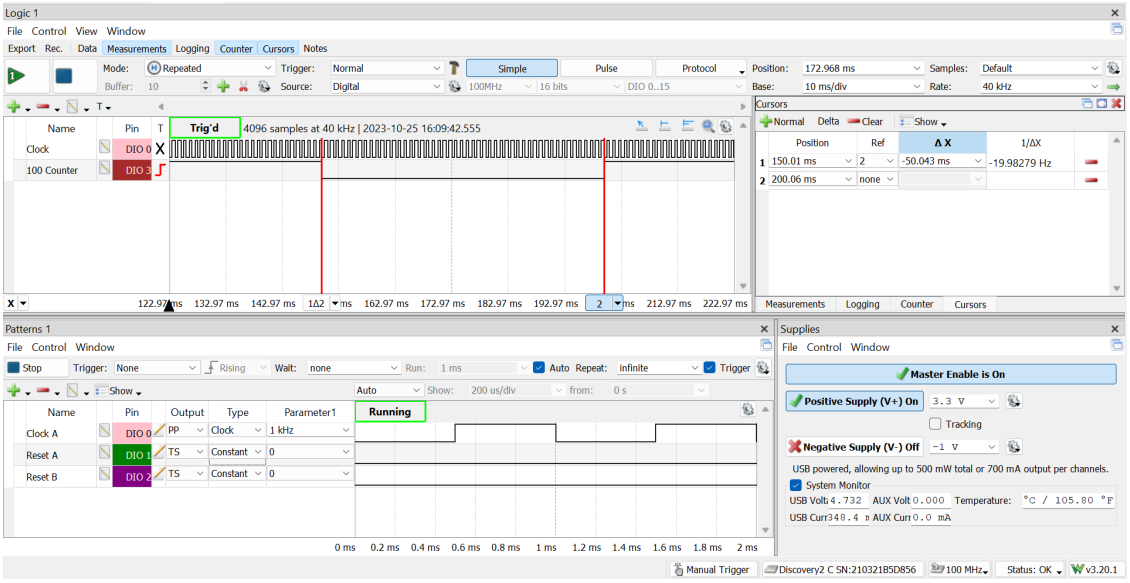


Figure 31: Buggy divide-by-100 counter (more of a divide-by-50 counter) output

[43]:
```
Image(filename = img_path("L5A_working divide by 100.png"))
```

[43]:

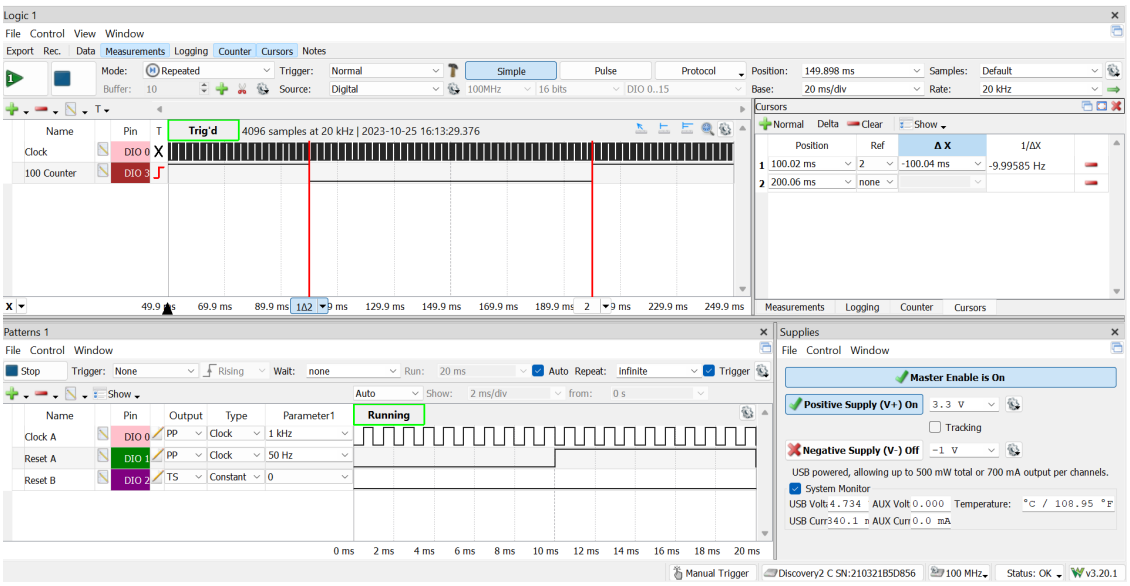


Figure 32: Working divide-by-100 counter output on Logic

4 Conclusion

4.0.1 Schmitt-Trigger Inverter

The inverter was my first introduction to logic in this course. I used a 74HC14A chip to invert an input signal; the chip responded to an input voltage and turned on when the voltage was below a certain threshold and off the voltage was above another threshold.

Again, I needed to find and read the datasheet for this component to understand how it worked. It's also interesting how the Schmitt-Trigger uses hysteresis to clarify its signals. In a sense, it acts as digital siganls, giving the input a threshold it must be between for the inverter to operate, instead of directly relying on the exact value of an input.

The inverter is probably also used widely in the industry as I can see how it can be quite useful in, well, 'inverting' signals. In essence, it's an if-else that's slightly more complicated than the signal from a direct source, since it actually does the 'opposite' of the signal. This could have applications in overcurrent protection or any other kinds of switches.

4.0.2 D-latch

Making a d-latch involved slightly more complicated understanding of logic gates. First, I found the documentation for the NAND gate chip we're using to confirm the NAND gates were working. I used my logical reasoning skills and understanding of how NAND gates operate (i.e. what inputs make the NAND gate output HI and vice versa) to do so.

I understood the the Data and Enable input pins of the d-latch controlled the output. In a nutshell, when enable is HI, Q reflects whatever state Data inputs, and when enable turns LO, Q stores whatever state Data was at before enable turned LO.

The d-latch also offered a basic way to store one bit data, explained in how Enable turning LO lets Q store the information in Data.

4.0.3 DAC

In this experiment, we used an R-2R resistive ladder to convert an 8-bit digital signal to an analog voltage.

I used my knowledge of how binary values are coverted to decimal/analog values to configure the digital input pins in such a way that the analog output voltage would increase linearly. As well, I understood how to determine the resolution of digital to analog coverter, since it has important ramifications to the limitations of the precision of the analog value.

Here, I learned how digital signals can be combined in a series to analog values using our knowledge of binary numbers and resistive circuits. This is an actual application of our theoretical knowledge of linear circuits and math, and is no doubt used often in the industry. While basic, I learned how simple the conversion from digital to analog signals can be, as well as how useful; instead of using analog inputs directly, we can take advantage of this system in case we have have digital (on-off)

signals and to minimize noise, since digital signals only have two states and are a lot les susceptible to noise than analog.

4.0.4 Counters

In the final part of the lab, I learned how we can use and combine digital counters to create more powerful counters.

In retrospect, it's possible I didn't create the counters entirely correctly as maybe each of the three bits for the ÷ by 5 counters shouldn't be wired together. Reasoning about how the bits in the counters store data and about how circuits work (if you connect two terminals with a wire they take on the same value) points towards this fact.

However, now I know how we can combine individual counters, and using WaveForm's logic, check that these resulting counters are operating correctly. We are also introduced to yet another way bits that can only store 0 and 1 values can be combined to store much more sophisticated, low-noise data. For example, the combined 8-bits of the ripple counter can act as a ÷ 100 counter and other digital 8-bits can do further operations such as timing.

[44]:
```
# @title
%%capture
!apt-get install texlive-xetex texlive-fonts-recommended␣
  ↪texlive-latex-recommended texlive-plain-generic pandoc
```

[45]:
```
# @title
# Capture to prevent lots of output... Remove this if troubleshooting!
%%capture

pdf_file_name = file_name.split('.')[0]+'.pdf'  # Same as file_name with .ipynb␣
  ↪changed to .pdf
!rm $file_name
!rm $pdf_file_name

import os

full_path = os.path.join(path, file_name)
!cp  "$full_path" ./

!jupyter nbconvert "$file_name" --to pdf
```