INTRODUCTION

A standard piece of equipment in an electronics lab is a *Function Generator*. A function generator creates variety waveforms (called functions) at variable frequencies.
In this lab you will build a function generator using a Pico W that you will use in future labs.

DESIGN OF FUNCTION GENERATOR

To make a function generator, first you need a way of creating continuous analog output from a program. This is done using a *DAC* (Digital to Analog Converter). There are many DAC breakout boards you can use with microcontrollers. However, if you program in MicroPython, the microcontrollers are too slow to make waveforms faster than about 2 kHz.

Fortunately several people in the huge community of microcontroller/MicroPython users came up with a very clever solution for the PicoW: An R-2R DAC ladder coupled with two special features of the PicoW, **`pio`** and **`dma`**.

The R-2R Ladder DAC

The R-2R ladder is a *parallel* device, which means that the data the microcontroller is sending has multiple, simultaneous digital data output lines. Figure 1 shows a 4-bit R-2R DAC. The simultaneous data input lines are **`D0`** – **`D3`**. Each of these can take on the values of either Vcc or 0. You will see in the analysis below that if you write a 4-bit binary number to **`D0`** – **`D3`**, you will get an analog value that varies from 0 to (near) Vcc.

One way to proceed is to use Kirchoff's Laws to the whole network and crank on the system of equations. But there is another way – using Thevenin's Theorem.

Thevenin's theorem: *Any network (or partial network) of voltage sources and resistors can be simplified to **one** voltage source in series with **one** resistor.*

You can also use Thevenin's theorem on a piece of a circuit, then successively on more and more of a circuit until you have the whole circuit analyzed.

Calculating The Thevenin Values

The circuit in Figure 1 is a schematic of an R-2R DAC. Each of the digital inputs, D0 – D3, can only take on the values 0 or V_{CC}. The output is an analog voltage from 0 to the maximum the DAC can output. Here are the rules for using Thevenin's theorem.

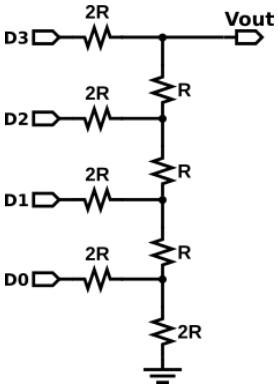


Figure 1: A 4-bit R-2R DAC ladder. A binary number applied to D0 - D3 creates a voltage between 0 and Vcc.

1. Identify the circuit or subcircuit you want to analyze. Identify the output.
   a) In this case we will use the whole circuit and the output is V_{OUT}.
2. **Find the Thevenin voltage**. If there is more than one voltage source, you will analyze them one at a time. For each one you set all of the other voltage sources to ground and calculate V_{OUT}.
   a) In this case we will start by grounding D0 – D2. This gives us the red circled circuit in Figure 2.
   b) In the space below redraw the circuit with the red subcircuit replaced by one equivalent resistor.

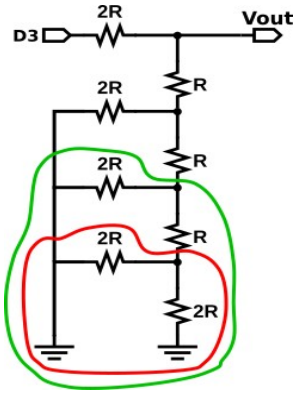


Figure 2: Start by setting D03 to Vcc, and the other digital inputs to ground. Analyze the subcircuit in red.

   c) Next with your new circuit, do the same with the part of the circuit in green in Figure 2. Draw the equivalent circuit below.

   d) Finally, do the same including the next two resistors. Draw your final circuit below. It should only have D1, two resistors and V_{OUT}.

   e) Finding V_{OUT} should now just be the output of a voltage divider. Below write the formula for V_{OUT} as a function of D3.

3. **Find the Thevenin Resistance**: Now that you have made the circuit simpler, it is easy to find the Thevenon resistance. Set D3 to ground and calculate the resistance from V_{OUT} to ground. In the space below draw the final Thevenin equivalent circuit. Label the voltage and resistance.

4. **Repeat for the other voltage sources in the original circuit**. Figure 3 shows how to redraw the circuit for analyzing D2. In the space below write the formulas for the equivalent voltage for sources D0 – D2.

5. **Use the principle of superposition to get the final answer**. *The magic of superposition*. Superposition states that the voltage across an element in a linear circuit is the algebraic sum of the voltage across that element due to *each* independent source acting alone. Since V_{OUT} is the voltage across the whole ladder, we can sum the contribution from each digital input. Write the formula for the full circuit below.

Summary

The final formula is

$$V_{OUT} = \frac{1}{2} D3 + \frac{1}{4} D2, + \frac{1}{8} D1 + \frac{1}{16} D0.$$

This is exactly what makes it a binary DAC. The output for 0xf (binary 0b1111) is $\frac{15}{16} V_{cc}$. The output for 0x0 is 0 V. Every integer from 0 to 0xf makes a voltage proportional to that number. In this case, there are 16 steps from 0 to $\frac{15}{16} V_{cc}$ in steps of $\frac{1}{16} V_{cc}$. The smallest step is called the LSD or *Least Significant Digit*.

Voltage & Time Digitization and DAC Resolution

This example is a 4-bit DAC. The voltage *steps* are 1/16th the total range of the DAC.

If we step the DAC from 0 to 0xf, we can easily see output voltage by looking at the waveform with an oscilloscope. Instead of a nice smooth curve, we see a series of steps going up and down. Each step is one LSB (Least Significant Bit) in size.

How can you do better? Add more inputs to the DAC.

The whole structure can be extended by adding four more R-2R sections and digital inputs to make an 8-bit DAC. This is what you are building in the DAC Project.

Why 8-bit? Because 8-bits gives you enough resolution to make a waveform that looks like a decent sine (or other type of) wave. I know sounds like a subjective judgment, but it comes from experience. You will see as you use the DAC that there are also “jumps” or “steps” in time called *time digitization*. The time digitization is set by how fast your microcontroller is, so a good engineering principle is to try to design your DAC so that the time and voltage digitization have about equal effects on the waveform generation.

BUILDING THE FUNCTION GENERATOR

The figure below is the schematic for the circuit of the Function Generator.

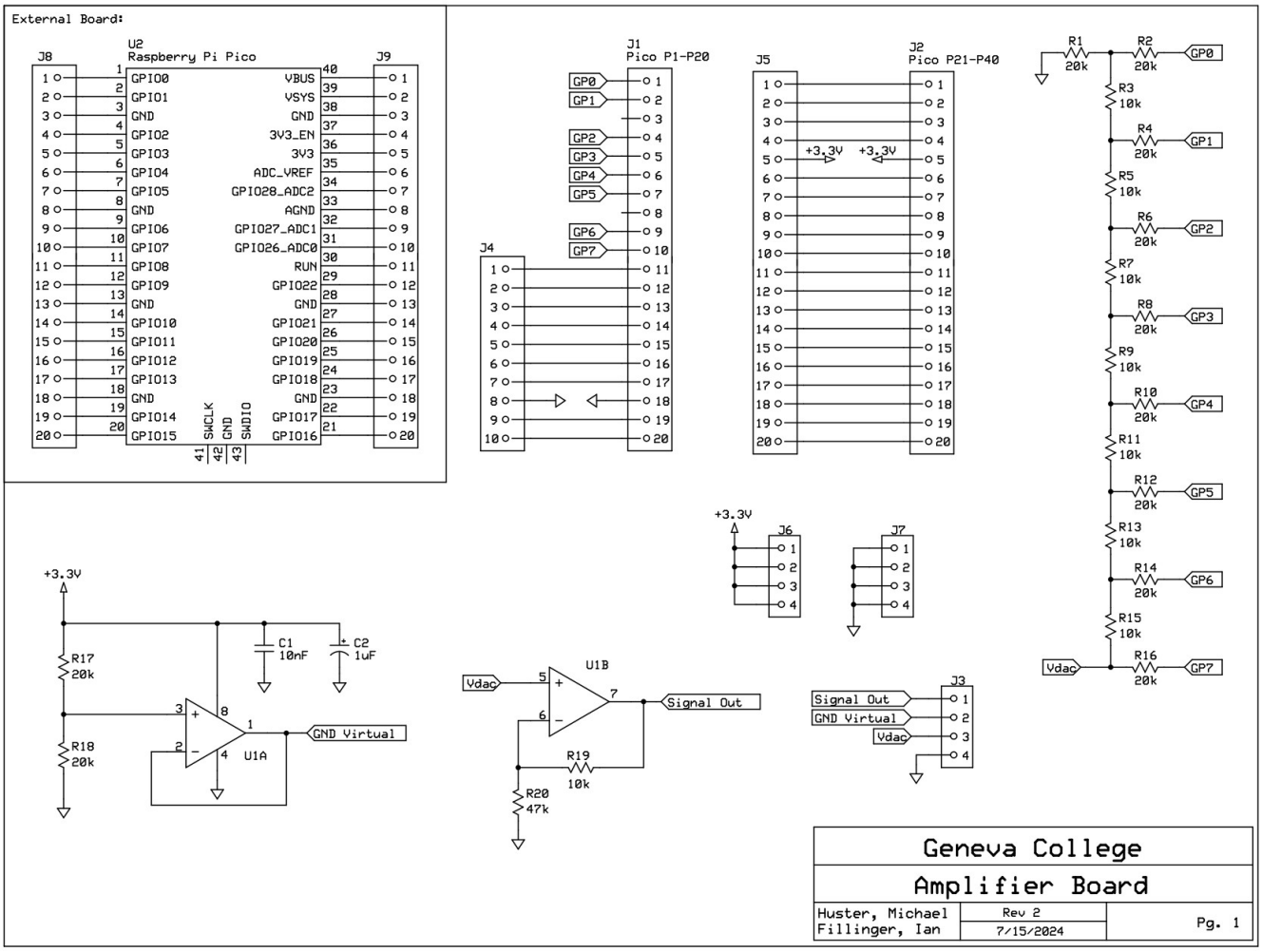
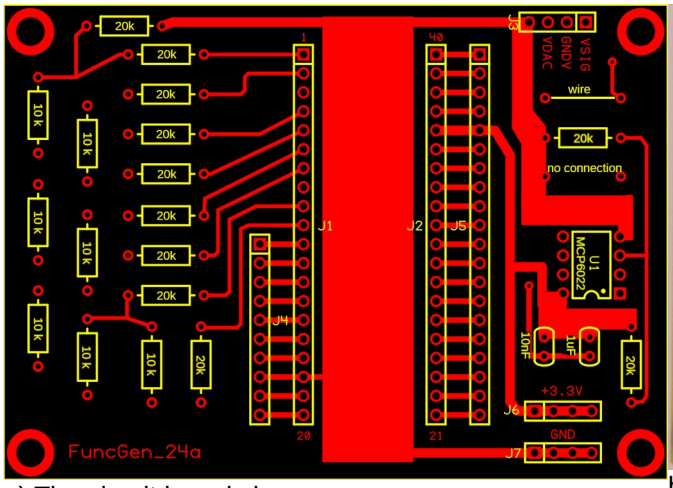


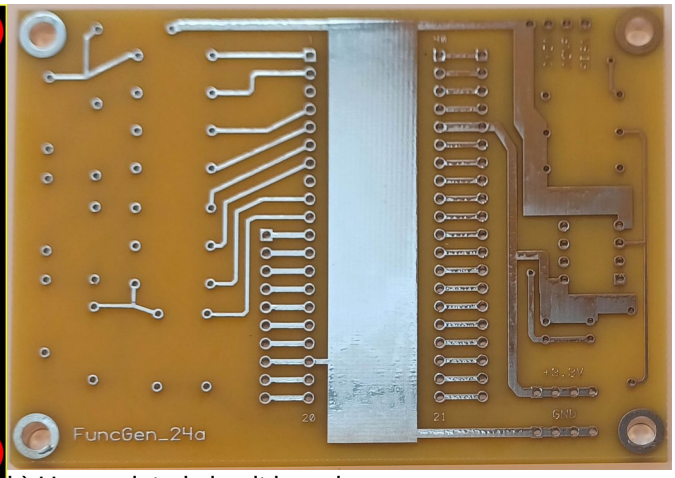
Figure 3: The schematic was drawn using the app PCB Express Pro. The app is free and encourages you to order the board from PCB Express.

The schematic has to *every detail* of the circuit right before proceding. I designed it for optional gain for op amp **`U1B`**. After testing, I decided it did not need gain so in the Soldering Guide I set **`R19`** to 0 by using a wire and left off **`R20`**.

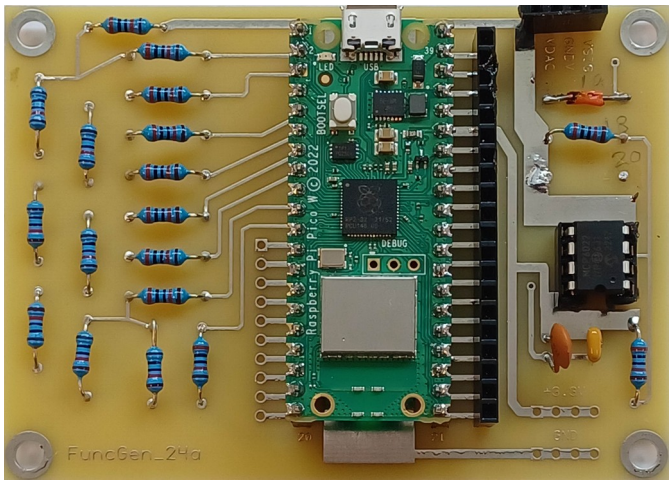
The figures below show the stages of building the Function Generator.



a) The circuit board view.



b) Unpopulated circuit board.



c) Finished circuit board except for some connectors.

The PCB (Printed Circuit Board) for the DAC is shown in Figure b above. All of the components will be mounted on this side, the front, of the PCB. The quick way to tell the front is that it has the board name, *FuncGen_24a* in the lower right corner. The Pico W and the op amp are *not* soldered to the board, but mounted in sockets. Figure c) shows a finished PCB, except for the three connector sockets not yet mounted.

It is hard to see the color codes on the resistors, so you can use a color code chart to select them. I strongly recommend you mount and solder all of the resistors of one value before do so with the other value.

Assembly.

- Start by mounting the resistors. *Pay close attention to the values!!!*
- Bend the leads so the wires can fit through the mounting holes.
- Put the wires through the holes and pull the wires from the opposite side until the resistor is close to the PCB.
- Flip the board over and solder the wires to the pad.
- Clip off the extra wire.
- I usually do two to four resistors at a time.
- For the Pico W sockets, plug the Pico W into the sockets then place the sockets in the PCB. Then solder them in place. After soldering, remove the Pico W.
- The op amp socket and connectors **`J4 - J7`** can be pushed through and soldered on the back. Try to keep them perpendicular to the circuit board.

Initial Testing

When you are finished, let the instructor inspect the board before you mount the Pico W and op amp.
I always try to check a board out before trusting it on a project. Here are some suggested steps.

1. Plug in the Pico W. *Make sure the USB connector is at the top of the board!*
2. Plug a USB cable from wall charger to the Pico W.
3. Feel the chips on the Pico W. If anything is hot unplug it immediately.
4. Unplug the Pico W, and plug the op amp into the board.
5. *Note*: Pin 1 of the op amp (and most chips) is marked with a small dot or depression by one of the corner pins, and sometimes a notch in the chip. You can see from the assembled board, the op amp is pointing toward the bottom of the circuit board.
6. Plug the board into charger and check the op amp. If it is hot, unplug the Pico W immediately.
7. If it passes these tests, plug the Pico W into a laptop.

TESTING THE FUNCTION GENERATOR

1. Load MicroPython on the Pico W as before.
2. Make a folder for storing the MicroPython code.
3. Download the file **`testDAC.py`** from the github page. Below is the code with my comments.

The first section of code imports some functions we need to run the DAC, Pin, ADC, and sleep. It also defines any parameters that set how the program works.

```
""" test_DAC.py
"""
print("test_DAC.py")

from machine import Pin, ADC
from time import sleep

N_LOOPS = 10
```

This part of the code defines the DAC pins and sets up the pins to read the voltages with the ADC.

```
# Set the output pins to the DAC
p7 = Pin(7, Pin.OUT)
p6 = Pin(6, Pin.OUT)
p5 = Pin(5, Pin.OUT)
p4 = Pin(4, Pin.OUT)
p3 = Pin(3, Pin.OUT)
p2 = Pin(2, Pin.OUT)
p1 = Pin(1, Pin.OUT)
p0 = Pin(0, Pin.OUT)
pins = [p0, p1, p2, p3, p4, p5, p6, p7]

# Set up the ADC to read the DAC output signal
adc_sig = ADC(Pin(26, Pin.IN))
adc_gndv = ADC(Pin(27, Pin.IN))
```

This function sets the DAC from a value. Since this is an 8-bit DAC, it sets the value to be less than or equal to 0xff (255). Using the & operator is a common way of limiting integers to a given number of bits. The code then sets the DAC to a starting value of 0; it turns the DAC off.

```
# Code to set DAC
```

```
def set_dac(x):
    # Make sure s is in range
    # The & is the "bitwise and" operator
    x &= 0xff
    # Set each pin
    for i_bit in range(8):
        # Fancy way of testing if bit is high
        if x & (1<<i_bit):
            pins[i_bit].high()
        else:
            pins[i_bit].low()

# Turn off the DAC
set_dac(0)
sleep(1)
```

This function reads the two ADC pins, converts them to voltages and returns both voltages.

```
# Function to read the signal out voltage
# and the virtual gound voltage
def adc_v():
    val_sig = adc_sig.read_u16()
    Vsig = 3.3 * val_sig / 0xffff
    val_gndv = adc_gndv.read_u16()
    Vgndv = 3.3 * val_gndv / 0xffff
    return Vsig, Vgndv
```

The code is fancier than just using the 8-bit mode. It also lets you test out the 4-bit mode. The 4-bit mode only uses pins **`D0`** – **`D3`**, like in the example above.

```
# Select 4 or 8 bit mode
answer = input("Test as a 4-bit or 8-bit DAC (Enter 4 or 8): ")
n_bits = int(answer)
# Make the default 8-bits
if n_bits != 4:
    n_bits = 8
print(f"Testing {n_bits} bit mode")
```

The number of steps depends on the number of bits you are using. The first line take the number 1 (the same in binary, decimal and hex) and shifts it in a binary fashion by **`n_bits`**. For 4-bits it will turn 1 into **`0x10000`** which is the decimal number 16.
Then the code loops over the full test **`N_LOOPS`** times.

```
n_steps = 1 << n_bits
# Repeat the measurements N_LOOP times
for i_loop in range(N_LOOPS):
```

The main test loops over every DAC step.

```
    # Loop over every DAC step
    for i in range(n_steps):
```

For the 8-bit test it outputs the number **`i`** to the DAC, then pauses for one tenth of a second. When you watch the DAC on an oscilliscope, it will look like a slowly ramping up voltage.

```
        # For 8-bits use every number
        if n_bits == 8:
            set_dac(i)
            sleep(0.1)
```

To work in the 4-bit mode, you want the first 4 bits of the number you send to the DAC to be zero, and you want to step the bits **`D4`** – **`D7`**. An easy way to do this is to shift **`i`** by 4 binary places with the **`<<`** operator. It then sleeps of 16 times longer than it does for the 8-bit mode so the sweep of voltages takes the same amount of time.

```
        else:
            # For 4-bit count by 16
            set_dac(i<<4)
            sleep(16 * 0.1)
```

Finally, print out the DAC value, the signal voltage, the virtual ground voltage, and the signal voltage relative to the virtual ground voltage for every step. When done, set the DAC back to 0 V.

```
        # Take data and print it
        (Vsig, Vgndv) = adc_v()
        print(f"{i:4d} {Vsig:7.4f} {Vgndv:7.4f} {Vsig - Vgndv:7.4f}")

set_dac(0)
print("D-O-N-E-!")
# EOF
```

Analyze the DAC Data

Connect an oscilloscope to the Function Generator. Clip one of the ground wires on a scope probe to a wire and put it in the GND socket. Clip wires on each of the scope probes and connect channel 1 to VSIG and channel 2 to GNDV. Set up the scope.

First test the 4-bit mode by running the program and entering **`4`**. You should see the voltage from the DAC slowly stepping up. In the Thonny *Shell* window, you should see four columns of data scrolling up.

To collect the data, first right-click in the *Shell* window and select *Clear*. Run the program. When it is done, in the *Shell* window type **`<Ctrl>A`** to select all, open a text editor window and type **`<Ctrl>V`** to paste it into the editor. Edit the file so the only lines in the file are the columns of numbers.

Start a jupyter notebook and write a program to read in the data using **`np.genfromtxt`**, and plot SIG versus DAC value. Fit the data to a line using **`np.polyfit`** and save the figure. For a perfect DAC, the data points would be exactly on the line. To calculate the quality of the DAC calculate the sum of (SIG – line)^2 and divide by the number of data points.

Show the graph and your quality number to your instructor before going on.

Once you have the analysis code working, repeat for the 8-bit DAC mode.

USING THE SINE GENERATOR

The function generator can actually make many kinds of periodic waves: sinusoids, triangle, sawtooth, pulses, etc. We don't need all that variety, so there is python code that only generates sine waves. Go to the github link above and download **`AWG_Sines.py`**.

Wire up the circuit in the Figure 4 to the right. Plug scope probe 1 into V_{SIG}, and probe 2 into V_{R2}, the voltage at the left of R2.

Answer below: What is the relationship of the signals V_{SIG}, and V_{R2}?

Draw a sketch of what you see on the scope for channels 1 and 2.

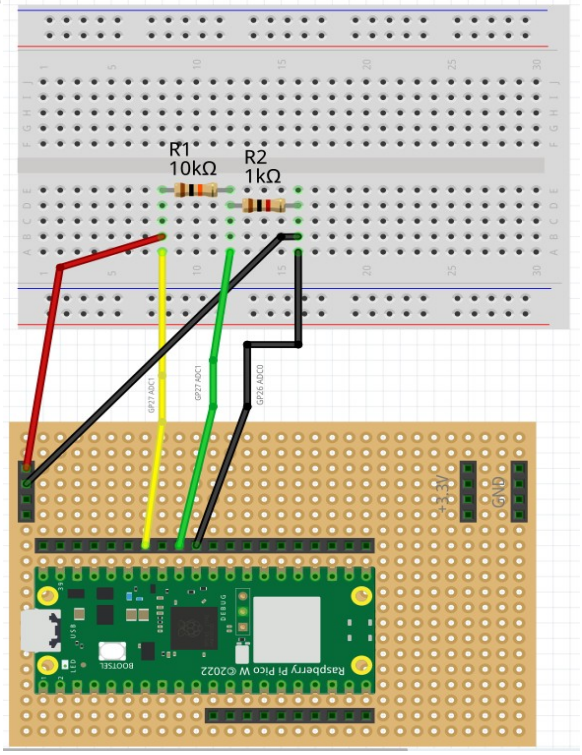


Figure 4: A "Fritzing" view of the circuit.

When you run the program:

- It asks if you want to space the frequencies linearly or logarithmically. If you press **`<Enter>`** it will default to linear spacing.
- For linear spacing you next enter **`f0`**, **`f1`**, **`nf`**, that is the starting and ending frequencies and the spacing between frequencies.
- For log spacing you enter **`f0`**, **`f1`**, **`nf`** where **`nf`** is the number of frequencies.
- The sine will have an amplitude of ½ V_{MAX} and be offset by ½ V_{MAX}, so on the oscilloscope the wave will fill 0 to 3.3V and be centered in the display.
- The program outputs a sleep time (you don't need it), the average value of V_{SIG}, the RMS of V_{SIG}, the RMS of V_{R2}, and the RMS of V_{R2}.