Lab 2: Analog & Digital Interfaces

Digital-to-Analog Converters (DACs) and Analog-to-Digital Converters (ADCs) are some of the most commonly used circuits today. They can be found in a wide variety of electronics, such as microcontrollers like the Arduinos we use in class, computers, phones, thermostats, etc.. This is because our computers and chips that process information from the real world, like temperature, must be stored **digitally**, while the information that we gather is itself an **analog** value. In the context of circuits, a digital value is one that is represented with 0s or 1s (a binary number), while an analog value can be anything in between (e.g. 0.5, 0.55555, etc).

Taking the example of a thermostat, in order to allow the CPU chip, which can only process binary numbers (sequences of 1s and 0s), that controls it to process the measured temperature, which is an analog value (like 20.2 °C or 75.9 °F), we need to have an ADC to convert that analog temperature into the digital value the chip is expecting. In order to have a phone call, an ADC must convert your voice (an analog voltage generated by the microphone) into a digital signal to be transmitted, and a DAC must take that digital signal and convert it back into an analog voltage for the speaker on the receiving end to play your voice. As you can see, DACs and ADCs play a very important role in the electronics we use everyday, and in this lab, we will be exploring how to build our own simple DAC and ADC.

Part 1: Digital-to-Analog Converters (DACs)

We will first build a 3-bit DAC to convert a **binary** input into an analog voltage, and then we will extend it to 4 bits using the knowledge you gained from building the 3-bit DAC. The binary input will come from the Arduino's digital I/O pins while the analog voltage will be probed with your Arduino and displayed on the serial monitor.

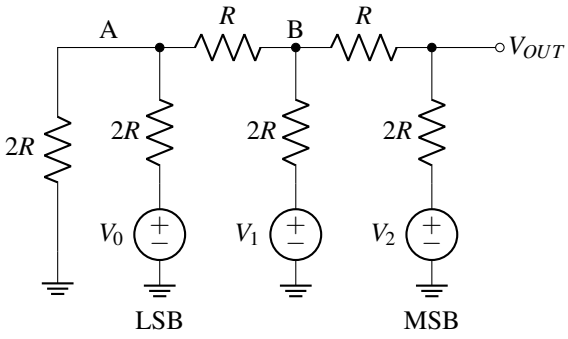
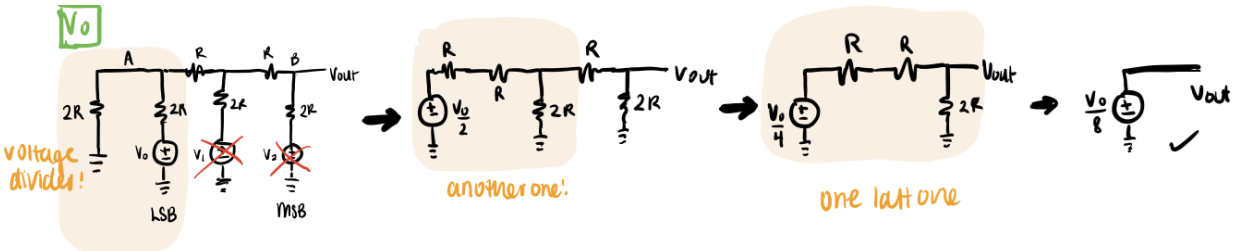


Figure 1: 3-bit DAC built from R-2R ladder

We can build a DAC using only resistors in a structure called the **R-2R ladder** (shown in Figure 1). This structure takes an n-bit binary input and converts it to an analog output voltage. The bits represent whether their corresponding voltage source is on (some reference voltage V_{ref}) or off (0V), where V_{ref} is 5V for the Arduino. In a previous homework, you have solved the R-2R ladder using KCL, nodal analysis, etc., so we will not show the full derivation of the circuit here. The general structure of the derivation involves analyzing the resulting output of each individual input voltage using Thevenin equivalent circuits, then summing all of the individual contributions to get V_{OUT} (superposition). We have done the first equivalent circuit for the LSB as an example.



Part 2: 4-Bit Analog-to-Digital Converter (ADC)

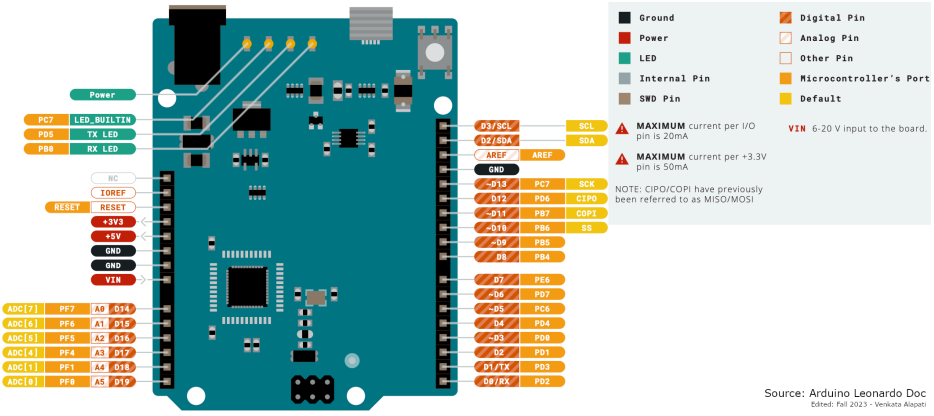
Given an analog voltage, an N-bit ADC converts it into an N-bit binary number in the digital domain. Digitization of a signal involves both **discretization**, i.e., we restrict its timesteps to being finitely small (infinitely small timesteps $\Rightarrow$ continuous time), and **quantization**, i.e., we fix a finite set of evenly spaced 2^N "states" (which can be converted to voltage values) that the signal can have at any given moment. In this case, this is the set of binary numbers between the decimal integer values of 0 and $2^N - 1$, inclusive.

There are many different schemes or algorithms to convert an analog voltage into a digital representation. Since we have a sorted, finite set of values, one of the conceptually simplest algorithms we could use is to run a linear search: start from 0 and count up, and stop once we find the closest match to the input analog voltage we're converting. As with any linear search, this is extremely inefficient, and there are many improvements we can make given that we have a sorted list of values, such as binary search. This leads into one of the most commonly used circuit architectures for analog-to-digital converters, the Successive Approximation Register ADC (SAR ADC), which is what we'll be building today using the 4-bit DAC from the previous part.

Brief Aside: Microcontrollers

In essence, a microcontroller is as a small "computer" containing memory, processors, and I/O (input/output) that execute a predefined program. Microcontrollers are the core of embedded systems and are often used to interact with/manage other devices. Some examples of microcontrollers include the Ardunio and ESP32 but don't include the popular Raspberry Pi which is consider to be more of a microprocessor. You can read more about the specific difference here.

The specific microcontroller we'll be using in the lab and for the rest of the class is the Arduino Leonardo. The documentation can be found on the Arduino website. The Arduino has numbered pins for I/O, which expect and output voltages between 0-5V and can be used as digital I/O for all pins and analog I/O for the pins labelled analog. Regarding power supply, you can employ the Arduino's VIN pin to provide power to the board, accepting input voltages ranging from 6 to 20 volts. In addition, the 3V3 and 5V pins are designed to output 3.3 volts and 5 volts, respectively. Lastly, GND serves as the grounding connection for the Arduino. For more information on the Arduino's pinout, please refer to the image below or here.



The SAR ADC Algorithm

The SAR ADC algorithm tries various binary "trial codes" by feeding them into a **DAC** to generate voltages and comparing the result with the analog input voltage using a **comparator**. It then uses feedback (**SAR logic**) to adjust the DAC voltage to get as close as possible to the analog voltage. The algorithm starts with the most significant bit (MSB), which is the bit with the largest binary weight (i.e. furthest to the left in a binary number). The circuit diagram of the SAR ADC is shown in Figure 2. The comparator outputs logic high (1) when $V_{IN} \geq V_{DAC}$ and logic low (0) when $V_{IN} < V_{DAC}$. An illustration of the algorithm is shown in Figure 3.

The concept the SAR-ADC can be illustrated by the game of guessing a number between 0 and $2^{20} - 1$, which is approximately 1 million. This is analogous to a game where you have to guess the number your friend is thinking of. They can only tell you if your guess is too high or too low, and you have 20 guesses to get as close as possible to your friend's number.

Would you start by guessing 0, then 1, then 2, and so forth? Or would you start in the middle, say at 500,000, check if you are too high or low, and then adjust to the next half, e.g., 250,000 or 750,000, depending on the result? The latter approach, known as divide-and-conquer, is a faster way to solve the problem.

The SAR ADC is essentially a circuit implementation of this game. The input voltage represents the "number" your friend has in mind. The Digital-to-Analog Converter (DAC) code is your guess, while the comparator gives the response (too high vs. too low). The SAR logic used to adjust the next code mirrors your strategy in deciding the next guess. For a 20-bit SAR DAC, you have 20 guesses; for an N-bit SAR DAC, you have N guesses.

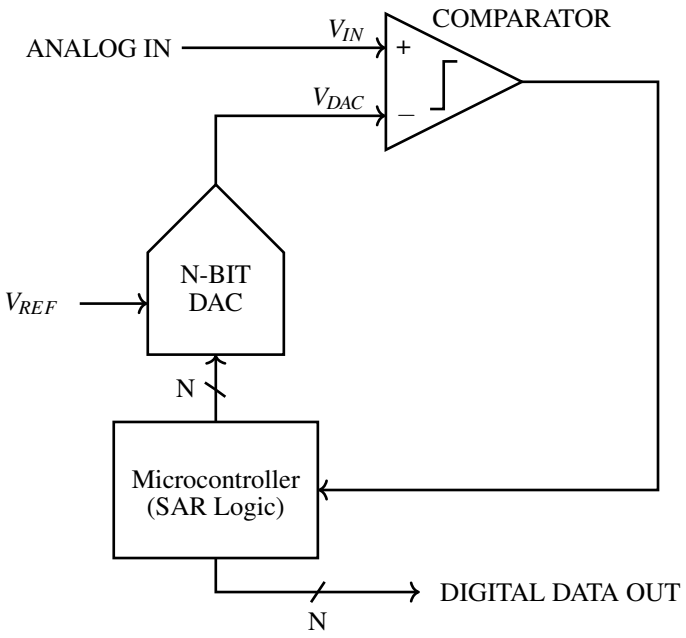


Figure 2: SAR ADC circuit diagram

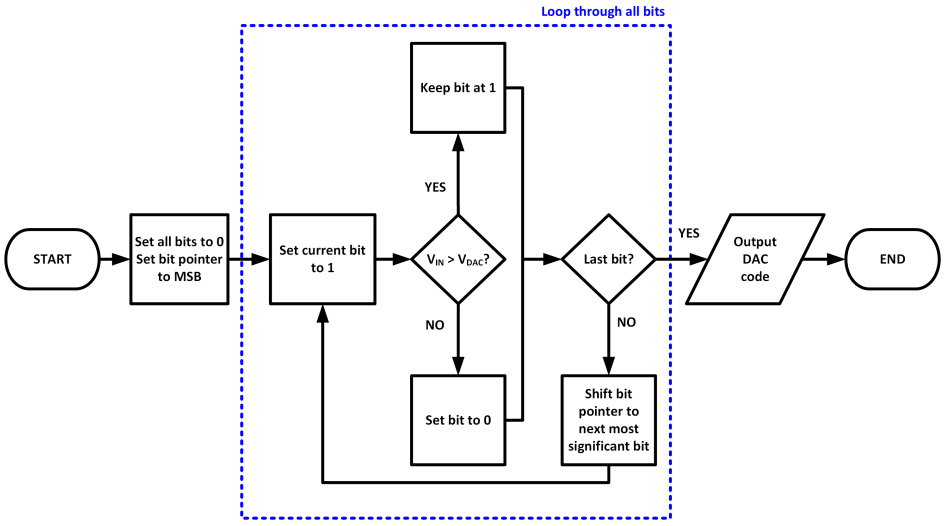
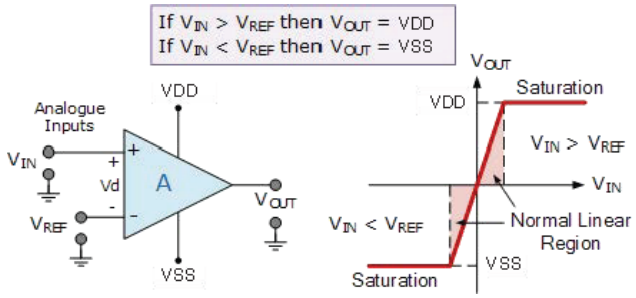


Figure 3: Flow chart of SAR ADC algorithm

The Comparator

A comparator is simply an op-amp configured in open-loop (no feedback), so that the output saturates at either the positive or the negative rail depending on whether the voltage at the noninverting (V^+) terminal of the op-amp is greater than or less than the voltage at the inverting (V^-) terminal. We are assuming the op-amp has sufficiently high open-loop gain (A) so that the "Normal Linear Region" in the graph is sufficiently narrow and thus is negligible.



You will use a comparator to compare the output of the DAC to the analog input voltage you get from the potentiometer.

Bringing It All Together

Let's walk through the operation of the N-bit SAR ADC shown in Figure 2 in detail.

The voltage we are trying to read is at the analog input. At the start of the algorithm, the SAR logic sets all bits in the register to zero, thereby setting all the DAC bit voltages to zero. At this point, $V_{DAC} = 0$.

Then, we begin trying DAC voltages. Just like in a binary search, we start with the MSB so that we can cut the search range in half. The SAR logic sets the MSB to 1, which turns on the corresponding bit in the DAC. V_{DAC} is then compared with V_{IN}, our analog input voltage. If $V_{IN} \geq V_{DAC}$, then the comparator's output is logic high (1), and the SAR logic just moves on to trying the next most significant bit. If $V_{IN} < V_{DAC}$, then the comparator's output is logic low (0), and the SAR logic sets the MSB back to zero before trying the next most significant bit. This cycle repeats until all bits in the register are set, at which point the digital data is sent out. This is the exact process shown in Figure 3.

Before moving to the 4-bit ADC that will be implemented in the lab, let's take a look at a 3-bit SAR ADC as an example. The voltage transfer curve of the 3-bit SAR ADC is shown in Figure 4, where $D_2D_1D_0$ is the output digital code of the ADC. Notice that the maximum voltage of the V_{DAC} is $\frac{2^N-1}{2^N}V_{REF}$ where N is the number of bits (N=3 in this case.) The Least Significant Bit (LSB) of the ADC can then be derived as $\frac{1}{2^N}V_{REF}$. For $V_{IN} = 0.3V_{REF}$, the operation of the SAR ADC algorithm is shown in Figure 5. The corresponding DAC code and comparator output in each cycle are also denoted in the timing diagram. **Note**: In this lab, the waveforms are different from those in Homework 2. That is because in the homework, we switch the current bit to the correct code (0 or 1, based on the output of the comparator) at the same time as we switch the next bit to the trial code (1). But in the lab, we switch the current bit to the correct code first and then switch the trial code after half a cycle.

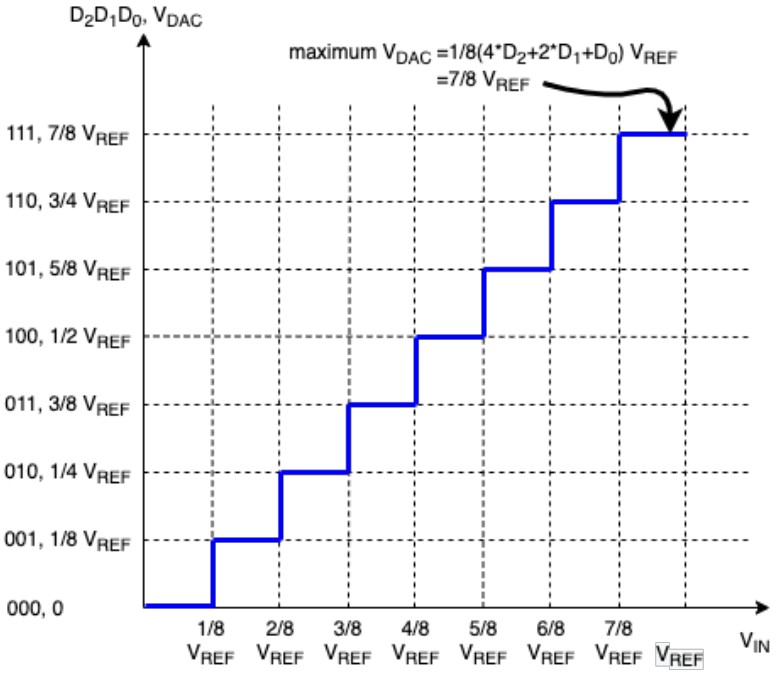


Figure 4: Voltage transfer curve of 3-bit SAR ADC

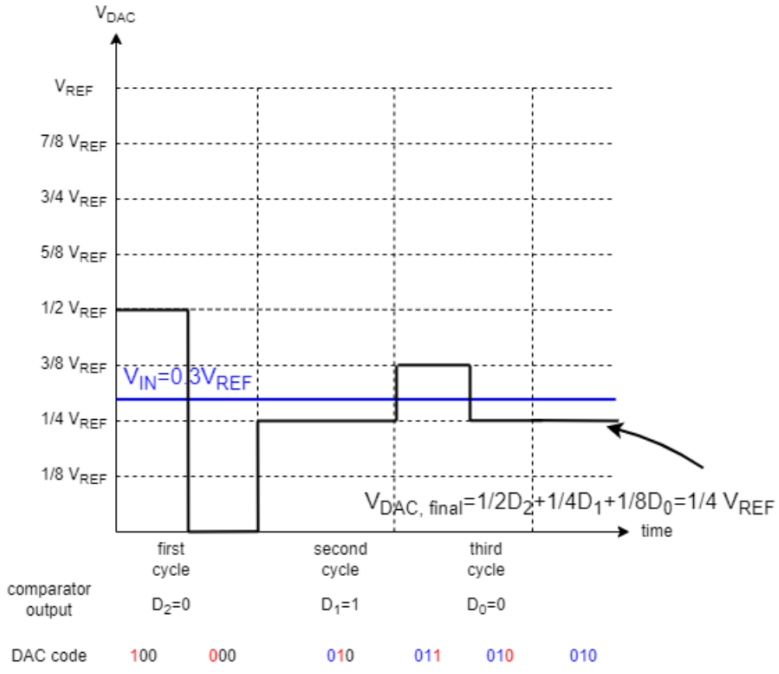


Figure 5: Timing diagram of SAR output code. The red digit in the DAC code is the bit being decided in the current conversion cycle; blue digits are the bits determined by the previous conversion cycles.

As another example, let's say that we are 5.8 hours behind in lecture (which we will represent as 5.8V), and we would like to convert that number into its digital representation so that we can store it easily. We have access to a 3-bit SAR ADC with a reference voltage V_{ref} of 8V. Figure 6 shows the output of the DAC in the ADC as the algorithm progresses and tries to find the closest match to the 5.8V input we feed it.

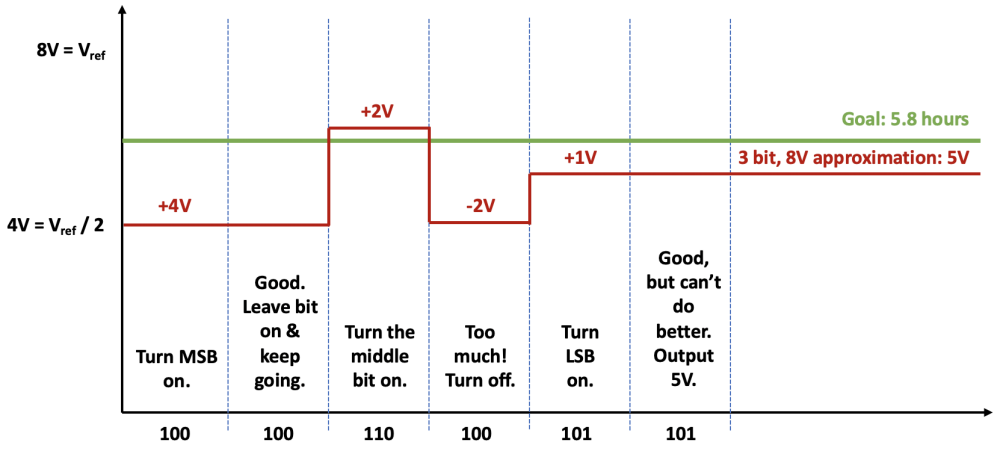
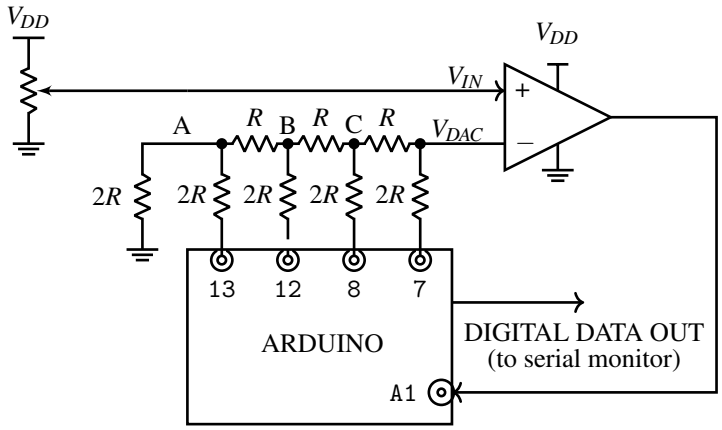


Figure 6: DAC output as SAR ADC algorithm progresses for an input of 5.8V

In the first step, you can see the MSB turns on and produces a DAC voltage of 4V, which is half of V_{ref}, as expected. The comparator tells us that this voltage is still less than the input voltage of 5.8V, so we leave this bit on. In the next step, we turn on the middle bit, which produces a step of 2V ($V_{ref}/4$). But now, the combined step of the MSB and middle step produces 6V, which is greater than our 5.8V input. The algorithm detects that we've overstepped and turns the middle bit off before proceeding onto the final bit, the LSB. In the final step, the LSB turns on, producing a step of 1V ($V_{ref}/8$), and the combined step of the MSB and LSB produces a 5V DAC output. The comparator once again tells us that this is less than the input voltage, but we're out of bits we can work with, so the algorithm terminates. The closest 3-bit digital representation we can get for this 5.8V is 101, which corresponds to an analog voltage of 5V. If we had more bits, we would be able to more closely approximate the true input voltage and get a better representation.

Below is a figure of how we will implement this ADC in the lab using our Arduino. Our ADC will have 4 bits of resolution, as shown in the image below. The Arduino handles the SAR logic (executed in code) and the 4 pins each represent one bit, each taking on a voltage of either 0V or V_{ref} (which is 5V for the Arduino).



Written by Mia Mirkovic (2019). Version 2.0, 2020.
Edited by Yi-Hsuan Shih, Steven Lu (2021). Version 3.0, 2021.
Edited by Megan Zeng (2022, 2023). Version 4.0, 2022.
Edited by Venkata Alapati, Junha Kim, Ryan Ma (2023). Version 5.0, 2023.