The University of Texas Rio Grande Valley
College of Engineering and Computer Science
Department of Electrical & Computer Engineering

ECEE 3225-03 Course name
Fall 2024

Lab Report 5

by

Jordan Lara

Enrique Casiano

Jesay Garcia

Instructor: Mr. Carlos Rodriguez Betancourth

October 30, 2024

Contents

I. INTRODUCTION

Purpose of the Experiment:

This experiment in ELEE 3225 Electrical Engineering Laboratory I seeks to thoroughly investigate the processes of digital-to-analog and analog-to-digital conversion. The primary objectives are to construct, test, and analyze the performance of both DACs and ADCs using various methods, such as resistor ladder networks and successive approximation techniques. Through these exercises, students will gain hands-on experience with signal conversion and learn how to program a microcontroller to approximate waveforms digitally. The overarching question addressed by this lab is: how can digital signals be accurately translated to analog forms, and conversely, how can analog signals be effectively represented digitally in an engineering context?

Background Information:

Digital-to-Analog Converters (DACs) and Analog-to-Digital Converters (ADCs) serve as critical interfaces between digital systems and the analog world. DACs convert binary data into analog voltage or current outputs, facilitating applications like audio output, video display, and signal modulation. A widely used implementation of DACs involves a resistor ladder network, specifically an R/2R ladder, which can create analog voltage outputs that are linearly proportional to the digital input code. For example, an 8-bit DAC receiving a digital input ranging from 0 to 255 will output a proportional voltage between 0 and 5V, where the resolution is determined by the number of bits in the digital input, resulting in incremental voltage steps (for an 8-bit DAC with 5V full-scale output, each step is approximately 0.0196V). ADCs perform the reverse operation: they take an analog voltage and convert it into a digital binary value. This lab introduces the successive approximation ADC, a type commonly found in microcontrollers due to its speed and efficiency. By utilizing a binary search method, successive approximation achieves a digital representation of an analog input within a set number of steps equal to the bit resolution of the converter (e.g., 10 steps for a 10-bit ADC). Additionally, the ADC's characteristics, such as resolution, accuracy, and speed, determine how well it can sample and convert rapidly changing analog signals into accurate digital representations. The PIC16F1517 microcontroller, used in this lab, has an internal 10-bit successive approximation ADC. This type of ADC, due to its design, is optimized for accuracy and quick conversion times, making it suitable for applications where high sampling rates and efficient data conversion are necessary. By configuring and testing this ADC, students will explore the internal registers responsible for ADC functionality, such as ANSEL, TRIS, ADCON0, and ADCON1, and measure performance through observing output on LEDs.

Hypothesis:

If the DAC and ADC are properly configured, constructed, and tested according to the principles of digital and analog conversion, the outputs should exhibit consistent linearity, resolution, and monotonicity in response to varied digital or analog inputs. This hypothesis assumes that accurate resistor values (within 1% tolerance) in the R/2R ladder will produce predictable voltage outputs for each digital input code in the DAC, and that the internal ADC of the PIC16F1517 will correctly approximate analog voltages according to its successive approximation process. Deviations in output linearity and accuracy are expected to arise from inherent component tolerances and resolution limits.

Summary of the Experiment's Content and Objectives:

This lab encompasses several stages, each aimed at reinforcing understanding of DAC and ADC fundamentals. First, students will manually construct a 4-bit DAC using an R/2R resistor ladder network to directly test output responses for a set of digital inputs. Next, they will connect this DAC to the microcontroller’s output port and program it to generate a staircase waveform, providing insight into DAC waveform synthesis capabilities. Following this, students will expand the circuit to an 8-bit DAC and program the microcontroller to create a smooth ramp waveform, allowing further exploration of waveform approximation. In parallel, students will work with the PIC16F1517’s internal ADC, configuring it to continuously sample analog inputs and display results on LEDs, enabling them to interpret how digital signals represent varying analog levels. For a final task, students will engage in one of two projects: synthesizing a specific waveform through the 8-bit DAC or constructing and testing a successive approximation ADC circuit, which will further illuminate the applications and limitations of DACs and ADCs in real-world signal processing. This lab builds a robust framework for comprehending the intricacies of signal conversion in electronic systems, equipping students with the skills needed to analyze performance metrics like resolution, offset error, and linearity in DACs and ADCs, which are integral to countless engineering applications. Through practical testing and theoretical grounding, students are expected to demonstrate a thorough understanding of DAC and ADC functionality, bridging the digital and analog domains crucial to modern electronics.

II. DESCRIPTION OF MAIN CONCEPTS

Key Terms and Definitions:

- Microcontroller:
  A microcontroller is a compact integrated circuit designed to govern a specific operation in an embedded system. Unlike microprocessors, microcontrollers include internal memory, timers, analog-to-digital converters, and input/output peripherals, making them ideal for control-oriented tasks.

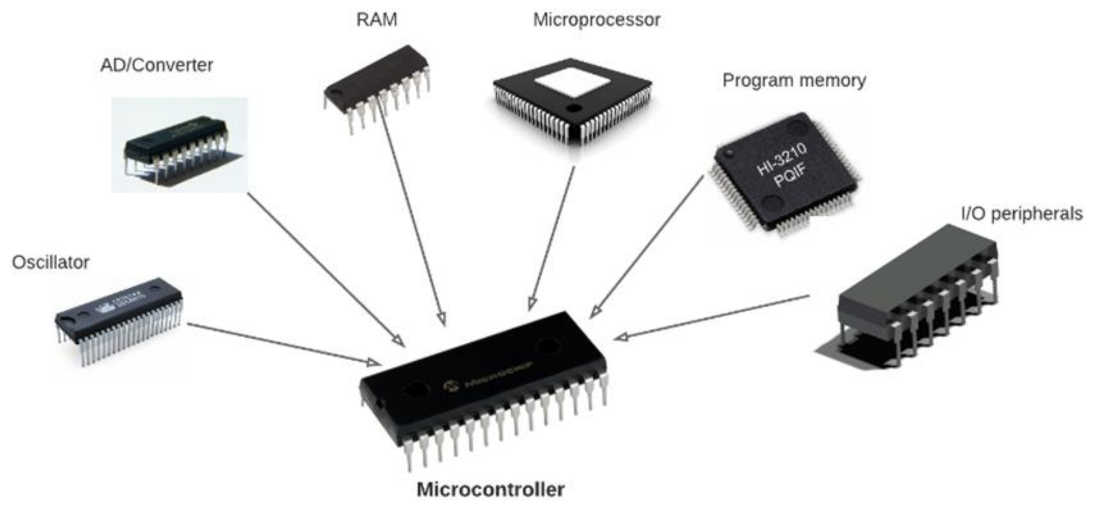


Figure 1: Microcontroller

- PIC16F1517 Microcontroller:
  The PIC16F1517 is an 8-bit microcontroller developed by Microchip, featuring 8192 words of flash memory, 512 bytes of RAM, and several specialized functions such as digital I/O, analog-to-digital conversion, and timers. It supports clock frequencies up to 20 MHz.

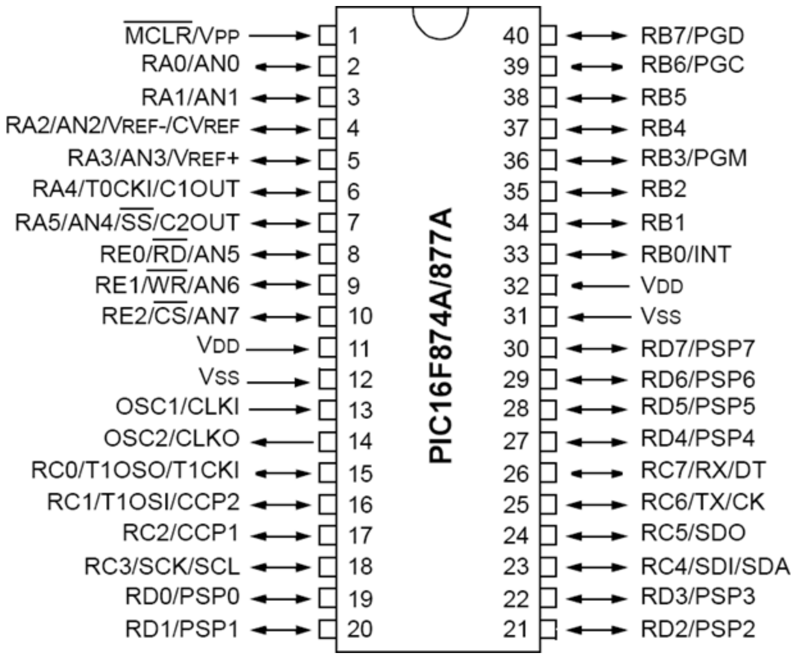
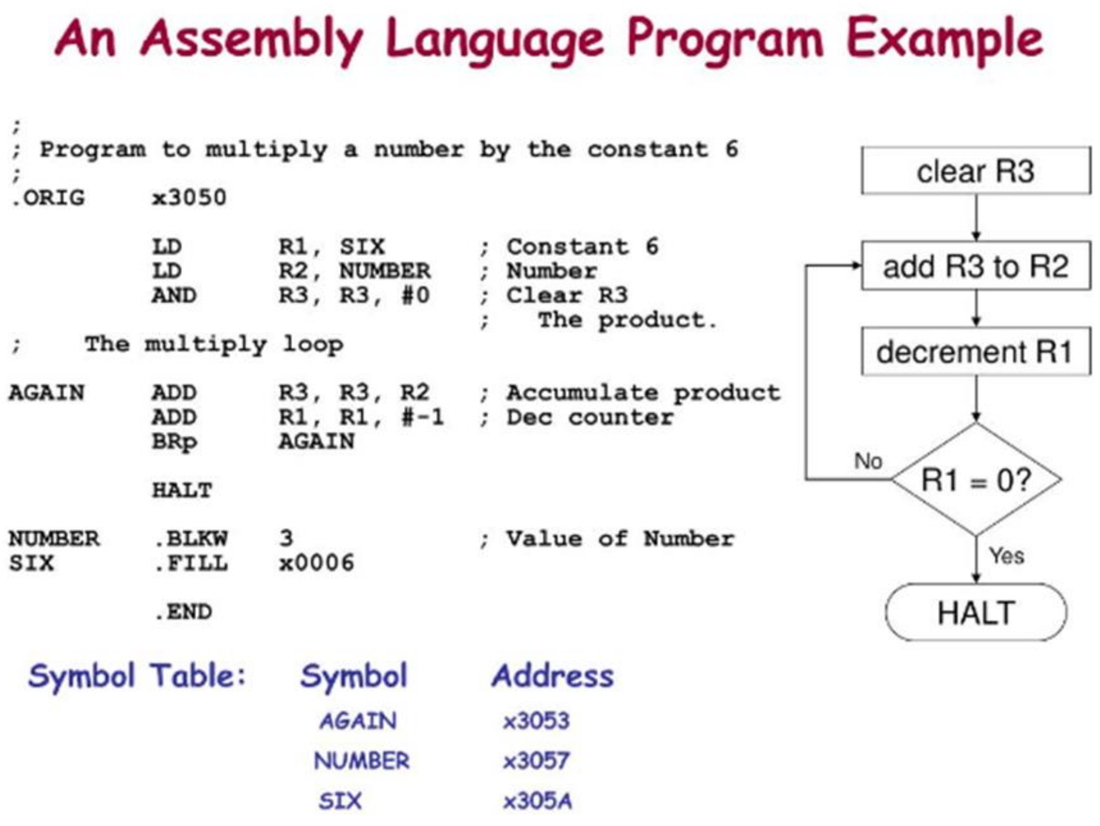


Figure 2: PIC16F1517

Figure 3: PIC16F1517

- Assembly Language:

  Assembly language is a low-level programming language where each instruction corresponds closely to the machine language of a computer or microcontroller. It provides fine control over hardware, making it suitable for time-critical operations.



An Assembly Language Program Example

```
;
; Program to multiply a number by the constant 6
;
.ORIG     x3050

          LD       R1, SIX        ; Constant 6
          LD       R2, NUMBER     ; Number
          AND      R3, R3, #0     ; Clear R3
                                  ;   The product.
;    The multiply loop

AGAIN     ADD      R3, R3, R2     ; Accumulate product
          ADD      R1, R1, #-1    ; Dec counter
          BRp      AGAIN

          HALT

NUMBER    .BLKW    3              ; Value of Number
SIX       .FILL    x0006

          .END
```

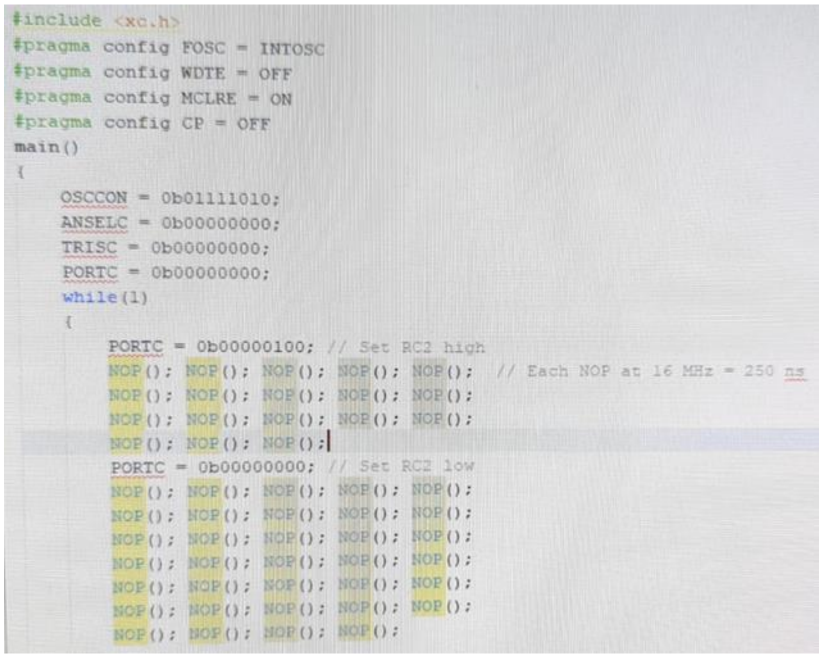
Symbol Table:

Symbol	Address
AGAIN	x3053
NUMBER	x3057
SIX	x305A

Figure 4: Assembly Language

- C Programming Language:

  C is a high-level programming language used to write more complex programs that are compiled into machine code. It is widely used in embedded systems due to its efficiency and control over hardware.



```c
#include <xc.h>
#pragma config FOSC = INTOSC
#pragma config WDTE = OFF
#pragma config MCLRE = ON
#pragma config CP = OFF
main()
{
    OSCCON = 0b01111010;
    ANSELC = 0b00000000;
    TRISC = 0b00000000;
    PORTC = 0b00000000;
    while(1)
    {
        PORTC = 0b00000100; // Set RC2 high
        NOP(); NOP(); NOP(); NOP(); NOP();  // Each NOP at 16 MHz = 250 ns
        NOP(); NOP(); NOP(); NOP(); NOP();
        NOP(); NOP(); NOP(); NOP(); NOP();
        NOP(); NOP(); NOP();
        PORTC = 0b00000000; // Set RC2 low
        NOP(); NOP(); NOP(); NOP(); NOP();
        NOP(); NOP(); NOP(); NOP(); NOP();
        NOP(); NOP(); NOP(); NOP(); NOP();
        NOP(); NOP(); NOP(); NOP(); NOP();
        NOP(); NOP(); NOP(); NOP(); NOP();
        NOP(); NOP(); NOP(); NOP(); NOP();
        NOP(); NOP(); NOP(); NOP();
```

Figure 5: C Programming Language

- **Potentiometer**:

  A potentiometer is a three-terminal variable resistor that allows for adjustable resistance within a circuit. By rotating its wiper, it varies the resistance between two points, making it useful for calibration, tuning, or as a voltage divider. Potentiometers are commonly used to control audio levels, brightness, or in applications requiring precise resistance adjustments.

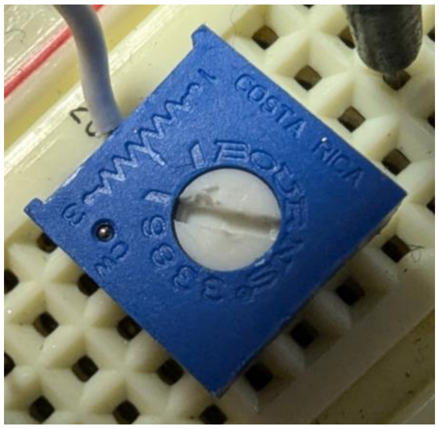


Figure 6: C Potentiometer

- **LED (Light-Emitting Diode)**:

An LED is a semiconductor light source that emits light when an electric current passes through it. Unlike traditional light bulbs, LEDs are energy-efficient, durable, and come in various colors. They are widely used as indicators in electronic devices, displays, and lighting due to their low power consumption and longevity.

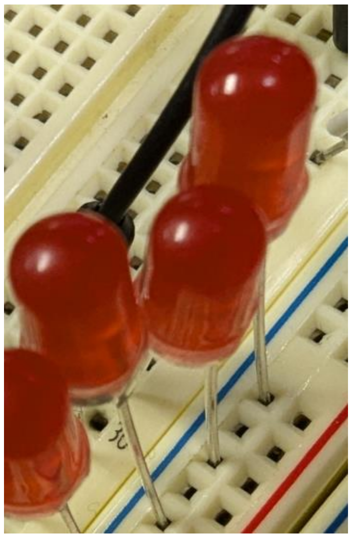


Figure 7: LEDs

- Breadboard:

  A breadboard is a solderless device used to build and test electronic circuits. In this lab, it is used to connect the microcontroller, motor, and other components without permanent connections.

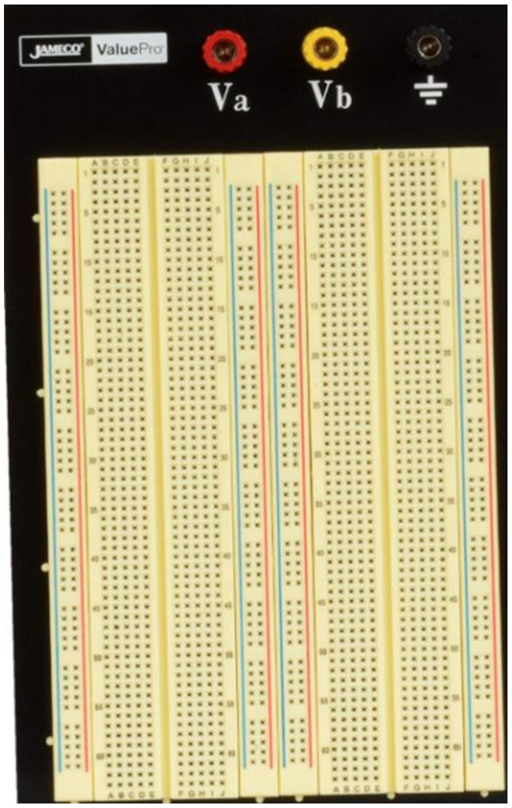


Figure 8: Breadboard

Theoretical Framework:

The principles governing DACs and ADCs are rooted in the conversion of continuous analog signals to discrete digital signals, a fundamental aspect of digital electronics. DACs operate by converting binary numbers into a proportional analog output. The R/2R ladder network employed in this lab is one of the simplest and most reliable methods for binary-weighted DACs, ensuring that each bit of the binary input code impacts the output voltage according to its binary significance. The use of resistors in the ratio R and 2R creates a scalable network, where each additional input bit adds another stage to the network.

Conversely, ADCs work by interpreting an analog signal and outputting its closest digital representation. The successive approximation method used in this lab involves a comparator and a DAC within the ADC. The comparator checks whether the DAC output (based on a digital guess) is higher or lower than the analog input, adjusting each guess through a binary search until it closely matches the input voltage. This method is efficient and enables the ADC to complete the conversion in a fixed number of steps, equal to the bit depth of the ADC.

In summary, this lab's approach to building and testing DACs and ADCs aims to provide a detailed exploration of how digital and analog signals are converted and interpreted. By analyzing the performance of DACs and ADCs, students gain practical knowledge on the accuracy, resolution, and limitations of these converters, essential in designing systems that require precise signal control and interpretation.

III. DEVELOPMENT

Materials and Methods

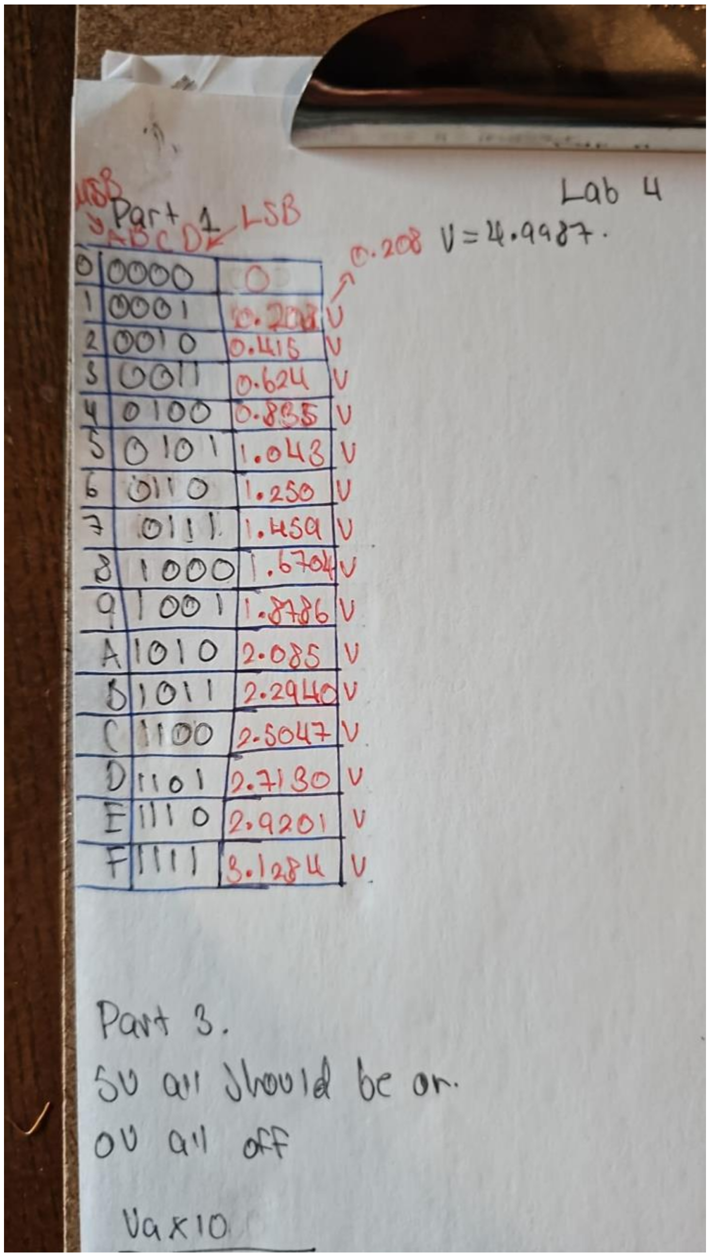
Materials:

For this experiment, we gathered the following equipment and components: a dual-trace oscilloscope, a digital multimeter, a logic analyzer (though it was optional), a power supply providing 5V, +12V, and -12V outputs, and a PIC microcontroller programmer (PicKit4). We also used a breadboard and assorted test cables to set up our circuits. For the actual components, we worked with a PIC16F1517 microcontroller, several resistors (ensuring they had 1% tolerance for accuracy), and a 0.1 µF capacitor.

Procedure:

PART 1:

Building and Testing the 4-Bit DAC: We started by constructing a 4-bit digital-to-analog converter (DAC) using an R/2R ladder network on our breadboard. We arranged resistors with values R and 2R in the configuration required, following the lab's instructions. After setting up the network, we connected each bit (A, B, C, D) to the voltage source, making sure A was the most significant bit (MSB) and D was the least significant bit (LSB). We connected the output of the DAC to the oscilloscope, which allowed us to monitor the voltage as we varied the binary inputs from 0000 to 1111. As we changed the input code in each possible combination, we recorded the output voltage corresponding to each input, noting any small deviations from theoretical values due to component tolerances.



Lab 4

MSB Part 1 LSB

0.208 V = 4.9987.

	ABCD	
0	0000	0
1	0001	0.208 V
2	0010	0.415 V
3	0011	0.624 V
4	0100	0.835 V
5	0101	1.043 V
6	0110	1.250 V
7	0111	1.459 V
8	1000	1.6704 V
9	1001	1.8786 V
A	1010	2.085 V
B	1011	2.2940 V
C	1100	2.5047 V
D	1101	2.7130 V
E	1110	2.9201 V
F	1111	3.1284 V

Part 3.

5V all should be on.

0V all off

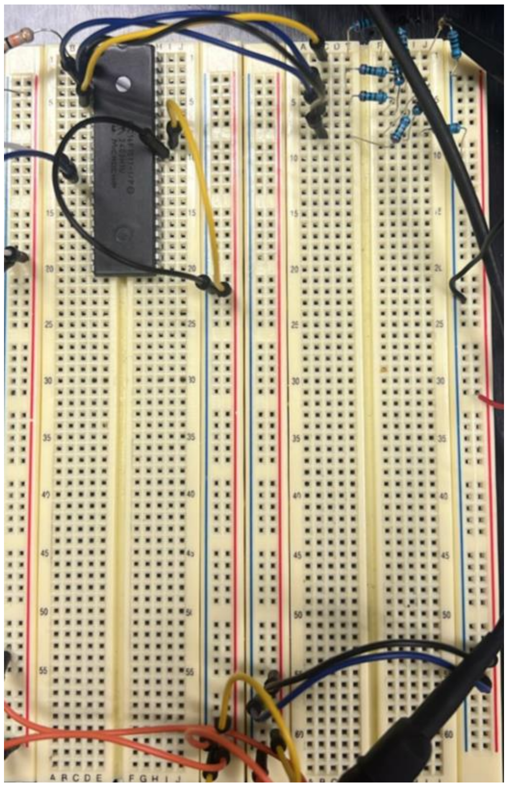
Va x10

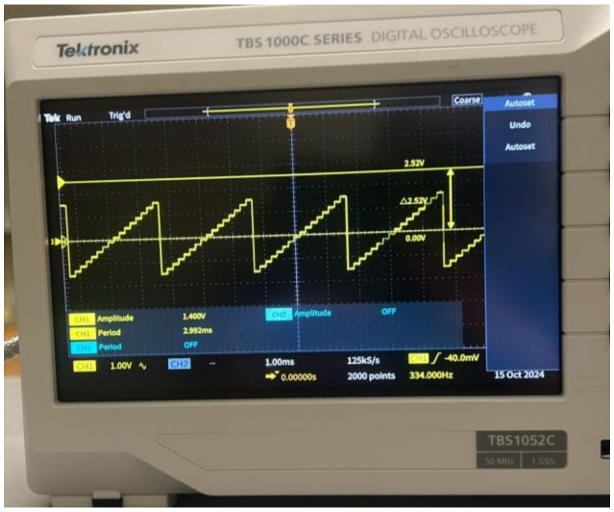
PART 2:

Programming the Microcontroller to Drive the DAC: Next, we connected our 4-bit DAC to a microcontroller output port, specifically PORTD, to automate the output sequence. We wrote a program that would cycle through a binary count from 0 to 15, which should produce a staircase waveform when viewed on the oscilloscope. After loading the code and running it, we observed a clear 16-step staircase waveform on the oscilloscope, confirming that the DAC worked as expected. We then decided to upgrade the DAC to 8 bits by adding additional R and 2R stages to the ladder network. Once this was done, we connected the 8-bit DAC to an 8-bit output port and modified our program to count from 0 to 255. Observing the waveform on the oscilloscope, we saw a smooth ramp function, which confirmed that the 8-bit DAC was performing as expected.

4-bit: Code and images

```
#include <xc.h>

// Configuration bits (adjust these for compatibility)

#pragma config FOSC = INTOSC   // Internal Oscillator

#pragma config WDTE = OFF      // Watchdog Timer Enable (WDT disabled)

#pragma config PWRTE = OFF     // Power-up Timer Enable (PWRT disabled)

#pragma config MCLRE = OFF     // MCLR Pin Function Select (MCLR pin function is
digital input)

#pragma config CP = OFF        // Program Memory Code Protection (disabled)

#pragma config BOREN = OFF     // Brown-out Reset Enable (disabled)

#pragma config LVP = OFF       // Low-Voltage Programming Enable (disabled)

#define _XTAL_FREQ 4000000 // Define oscillator frequency

void main(void) {

   // Configure PORTB

   TRISA = 0xF0; // Set RB0-RB3 as outputs, RB4-RB7 as inputs

   PORTA = 0x00; // Clear PORTB initially

   unsigned char count = 0;

   while(1) {

      PORTA = (PORTA & 0xF0) | count; // Output count on RB0-RB3

      __delay_us(10); // Delay to observe each count value

      count = (count + 1) & 0x0F; // Increment count with wrap-around after 15

   }

}
```

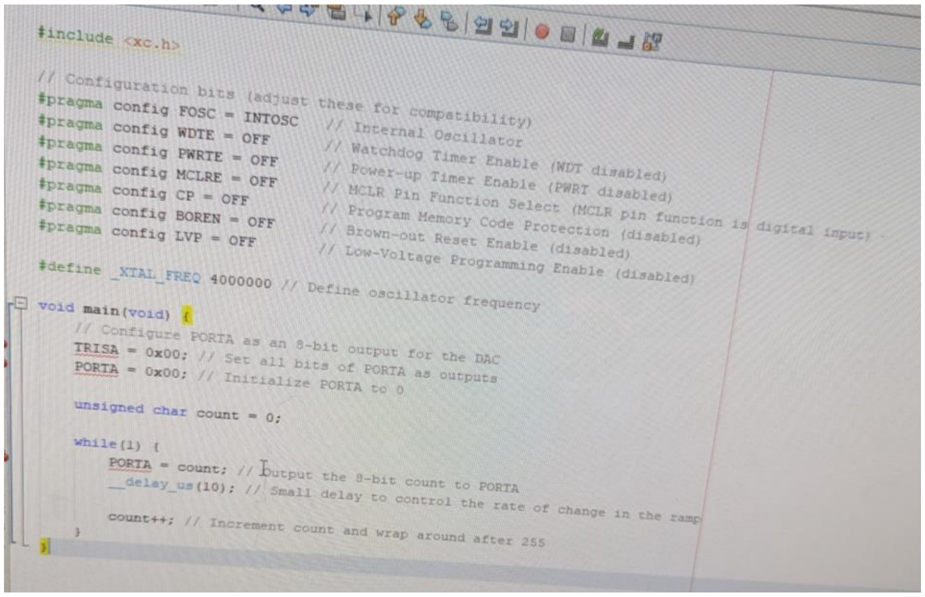




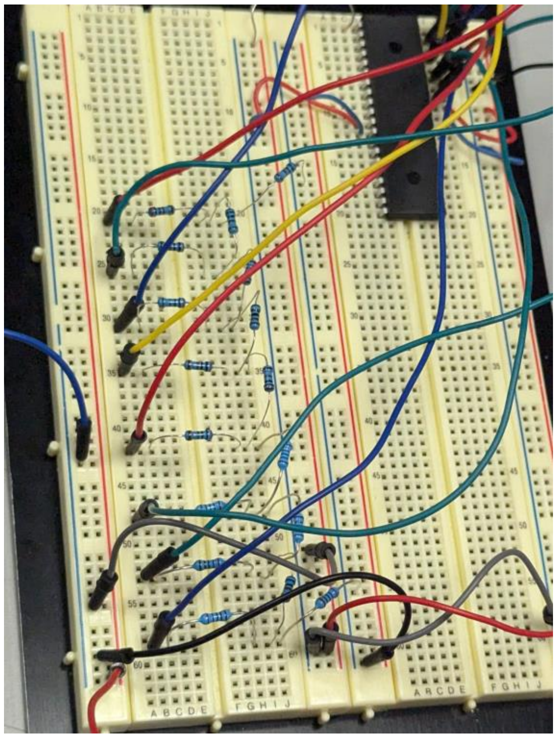
8-bit: Code and Images

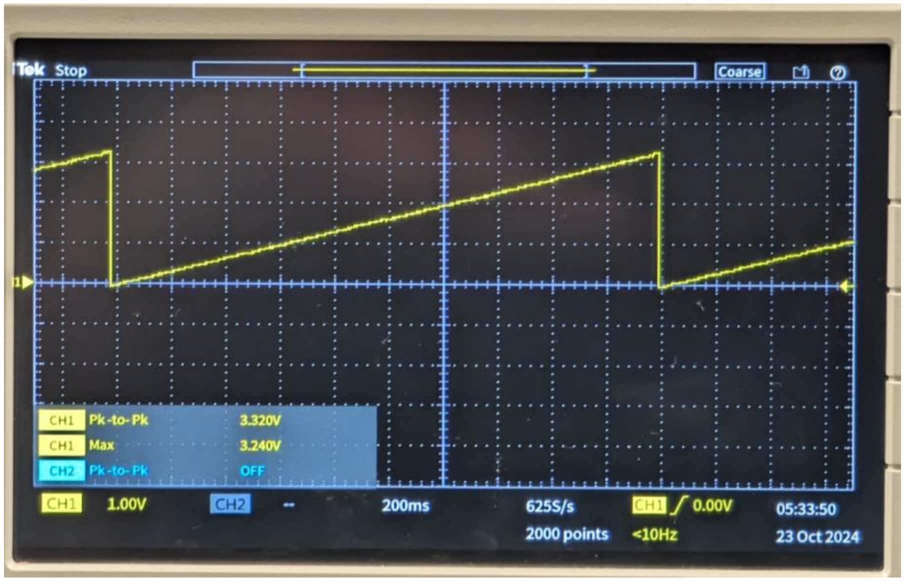
```
#include <xc.h>


// Configuration bits (adjust these for compatibility)
```

```
#pragma config FOSC = INTOSC    // Internal Oscillator

#pragma config WDTE = OFF       // Watchdog Timer Enable (WDT disabled)

#pragma config PWRTE = OFF      // Power-up Timer Enable (PWRT disabled)

#pragma config MCLRE = OFF      // MCLR Pin Function Select (MCLR pin function is
digital input)

#pragma config CP = OFF         // Flash Program Memory Code Protection (disabled)

#pragma config BOREN = OFF      // Brown-out Reset Enable (disabled)

#pragma config LVP = OFF        // Low-Voltage Programming Enable (disabled)

#define _XTAL_FREQ 4000000      // Define oscillator frequency

void main(void) {

   // Configure PORTA as an 8-bit output for the DAC

   TRISA = 0x00;               // Set all bits of PORTA as outputs

   PORTA = 0x00;               // Initialize PORTA to 0

   unsigned char count = 0;

   while(1) {

      PORTA = count;          // Output the 8-bit count to PORTA

      __delay_us(10);         // Small delay to control the rate of change in the ramp

      count++;                // Increment count and wrap around after 255

   }

}
```



```c
#include <xc.h>

// Configuration bits (adjust these for compatibility)
#pragma config FOSC = INTOSC    // Internal Oscillator
#pragma config WDTE = OFF       // Watchdog Timer Enable (WDT disabled)
#pragma config PWRTE = OFF      // Power-up Timer Enable (PWRT disabled)
#pragma config MCLRE = OFF      // MCLR Pin Function Select (MCLR pin function is digital input)
#pragma config CP = OFF         // Program Memory Code Protection (disabled)
#pragma config BOREN = OFF      // Brown-out Reset Enable (disabled)
#pragma config LVP = OFF        // Low-Voltage Programming Enable (disabled)

#define _XTAL_FREQ 4000000 // Define oscillator frequency

void main(void) {
    // Configure PORTA as an 8-bit output for the DAC
    TRISA = 0x00; // Set all bits of PORTA as outputs
    PORTA = 0x00; // Initialize PORTA to 0

    unsigned char count = 0;

    while(1) {
        PORTA = count; // Output the 8-bit count to PORTA
        __delay_us(10); // Small delay to control the rate of change in the ramp

        count++; // Increment count and wrap around after 255
    }
}
```



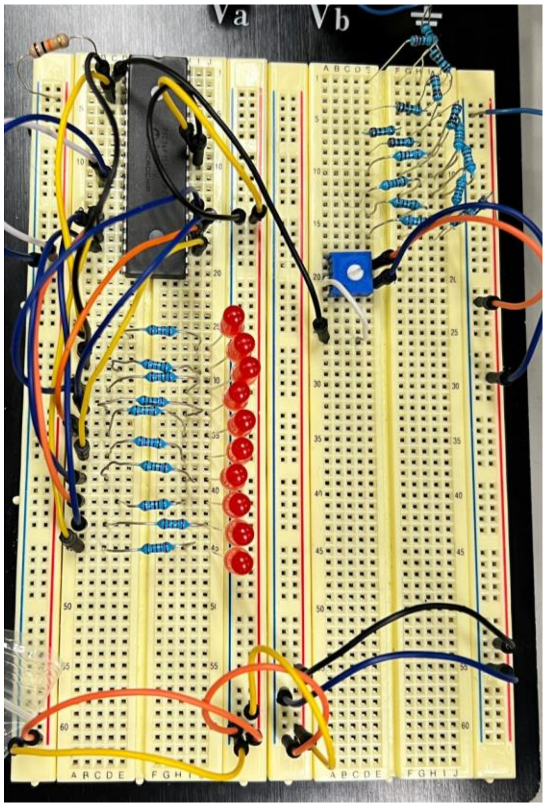


PART 3:

Configuring and Testing the ADC: Our next focus was on testing the analog-to-digital converter (ADC) built into the PIC16F1517 microcontroller. We began by configuring the ADC registers (ANSEL, TRIS, ADCON0, and ADCON1) to set the correct analog input, choose right justification for results, and use VDD as the reference voltage. We wrote a simple program to sample an analog input voltage and display the digital output on a row of LEDs, which allowed us to verify the conversion visually. To test the ADC's accuracy, we adjusted the analog input in 0.5V increments, noting the corresponding output in binary. We also recorded the minimum input voltage needed to produce an all-ones output on the ADC, finding it was slightly below the theoretical maximum of 5V.

ADC Code and images:

```
#include <xc.h>

// Configuration bits
#pragma config FOSC = INTOSC   // Internal Oscillator
#pragma config WDTE = OFF      // Watchdog Timer Enable (WDT disabled)
#pragma config PWRTE = OFF     // Power-up Timer Enable (PWRT disabled)
#pragma config MCLRE = OFF     // MCLR Pin Function Select (MCLR pin function
is digital input)
#pragma config CP = OFF        // Program Memory Code Protection (disabled)
#pragma config BOREN = OFF     // Brown-out Reset Enable (disabled)
#pragma config LVP = OFF       // Low-Voltage Programming Enable (disabled)

#define _XTAL_FREQ 4000000 // Define oscillator frequency

void setup_adc(void) {
    TRISA = 0x01;    // Set RA0 as input, rest as output
    ANSELA = 0x01;   // Set RA0 as analog, rest as digital

    // Set up ADC:
    ADCON0 = 0b00000001; // Enable ADC, select AN0 (RA0)
    ADCON1 = 0b10000000; // Right justified, use VDD (5V) as reference
}

unsigned int read_adc(void) {
    ADCON0bits.GO = 1; // Start ADC conversion
    while(ADCON0bits.GO); // Wait for conversion to complete
    return ((ADRESH << 8) + ADRESL); // Return the 10-bit ADC result
}

void main(void) {
    setup_adc();

    TRISA = 0x01; // RA0 as input, RA1 and RA2 as output for MSBs
    TRISC = 0x00; // All PORTC as output for LSBs
    ANSELC = 0x00; // All PORTC pins as digital

    while(1) {
        unsigned int adc_result = read_adc(); // Read ADC value from 0 to 1023

        // Directly set PORTC to the LSBs (lower 8 bits) of the ADC result
        PORTC = adc_result & 0xFF;

        // Directly set RA1 and RA2 based on the 2 MSBs of the ADC result
        RA1 = (adc_result & 0x100) ? 1 : 0;  // Check bit 8
        RA2 = (adc_result & 0x200) ? 1 : 0;  // Check bit 9

        // Simple delay for stability
        for(unsigned int i = 0; i < 500; i++) {
            __delay_us(200);
        }
    }
}
```



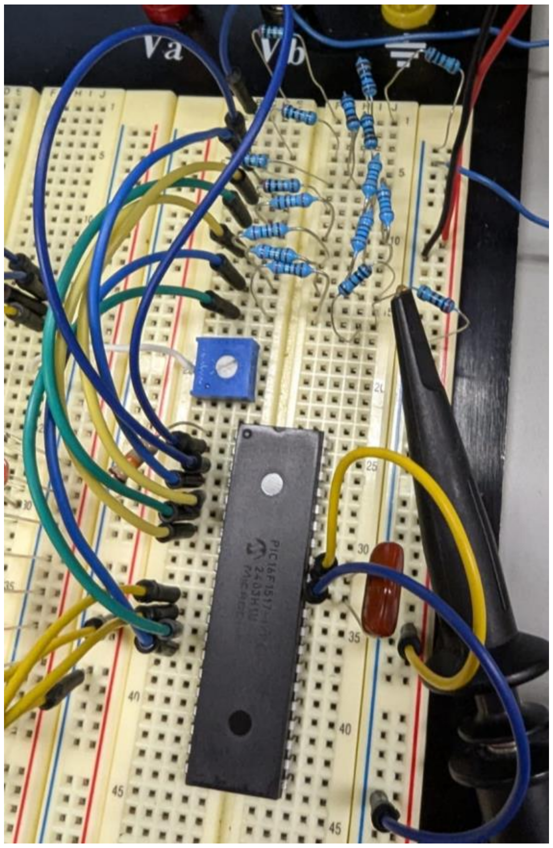
PART 4:

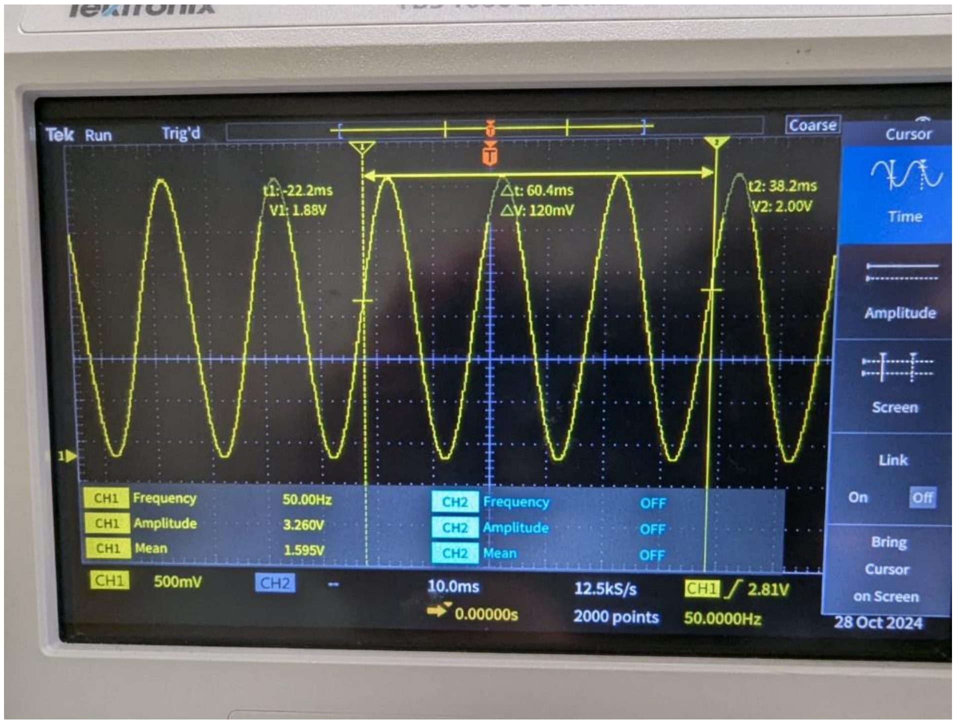
Project Work – Waveform Synthesis: For the final part of our lab, we chose Project A, which involved using our 8-bit DAC to synthesize a specific waveform. We programmed the microcontroller to output a waveform pattern as assigned by our instructor, then verified the result by observing it on the oscilloscope. Although the output closely resembled the expected waveform, we noticed slight discrepancies, likely due to the microcontroller's limited processing speed and the inherent limitations in DAC resolution.

Waveform Synthesis: Code and Images

```
#include <xc.h>
#include <math.h>  // Include math library for sine calculations

// Configuration bits
#pragma config FOSC = INTOSC   // Internal Oscillator
#pragma config WDTE = OFF      // Watchdog Timer Enable (WDT disabled)
#pragma config PWRTE = OFF     // Power-up Timer Enable (PWRT disabled)
#pragma config MCLRE = OFF     // MCLR Pin Function Select (MCLR pin function
is digital input)
#pragma config CP = OFF        // Program Memory Code Protection (disabled)
#pragma config BOREN = OFF     // Brown-out Reset Enable (disabled)
```

```
#pragma config LVP = OFF       // Low-Voltage Programming Enable (disabled)

#define _XTAL_FREQ 32000000    // Correct clock frequency in Hz (3.25 MHz)
#define NUM_POINTS 82         // Number of points in one sine wave cycle

// Sine wave array, generated dynamically based on the sine function
unsigned char sine_wave[NUM_POINTS];

unsigned char wave_index = 0;

void generate_sine_wave(void) {
   for (int i = 0; i < NUM_POINTS; i++) {
      sine_wave[i] = (unsigned char)((127.5 * (1 + sin(2 * M_PI * i / NUM_POINTS))));
   }
}

void setup_dac(void) {
   // Configure PORTA as output for 8-bit DAC
   TRISA = 0x00;  // All PORTA as output
   ANSELA = 0x00; // Ensure PORTA is digital
}

void output_waveform(void) {
   // Output the current sine wave value to the DAC (PORTA)
   PORTA = sine_wave[wave_index];

   // Increment wave_index to move to the next value
   wave_index++;
   if (wave_index >= NUM_POINTS) {
      wave_index = 0;  // Loop back to the beginning of the waveform
   }
}

void main(void) {
   setup_dac();        // Setup DAC for output
   generate_sine_wave();  // Generate sine wave values dynamically

   while(1) {
      output_waveform();  // Output waveform value to DAC

      // Delay for 78µs to maintain 50Hz frequency (adjusted for new clock
frequency)
      //__delay_us(78);  // Adjusted delay for 50Hz (78µs per step)
  }
}
```


Va
Vb
PIC16F1517



Data Analysis

Analysis of 4-Bit DAC:

The measured output voltages for each binary input in our 4-bit DAC showed close alignment with theoretical values, and the small deviations we noted were primarily due to resistor tolerances. The offset and full-scale error were minimal, validating the accuracy of our DAC within acceptable limits for a 4-bit resolution.

Analysis of 8-Bit DAC:

The 8-bit DAC performed as expected, with a step size of about 0.0196V. The ramp waveform was smooth and consistent, though we observed slight variations that might have been due to timing delays from the microcontroller or slight differences in resistor values.

ADC Analysis:

The ADC accurately converted analog inputs to digital outputs, with small offset errors appearing at the lower voltage levels, possibly due to slight inaccuracies in the ADC configuration or inherent microcontroller limitations. The measured full-scale voltage, slightly below 5V, could be due to minor variances in the reference voltage.

Project A (Waveform Synthesis):

When synthesizing the waveform using the 8-bit DAC, we achieved a pattern close to the instructor’s assigned waveform, although there were slight discrepancies due to the microcontroller’s processing limitations. Despite this, the output was largely successful and showed the DAC’s ability to produce analog signals from programmed digital values.

IV. CONCLUSIONS

Summary of Findings:

In this experiment, we successfully completed Project A, synthesizing a stepped sine waveform using the DAC on the PicKit4 microcontroller. By programming the microcontroller to output sequential binary values, we created an analog sine wave approximation observed on the oscilloscope. The 4-bit and 8-bit DAC configurations provided expected voltage step changes for each binary input, and the ADC configuration on the PIC16F1517 accurately converted analog inputs to digital outputs.

Interpretation:

The experiment demonstrated that the DAC could produce a reliable stepped approximation of an analog waveform, illustrating its capacity to bridge digital programming with analog output. The accuracy of the DAC’s output waveform aligned well with the theoretical predictions, with minimal deviations due to component tolerances. The ADC’s successive approximation method effectively captured varying analog inputs as binary values, supporting the hypothesis that both DAC and ADC would perform reliably with a well-designed setup. This successful waveform synthesis confirmed the feasibility of using microcontroller-driven DACs for analog signal generation in real-time applications, reinforcing the theoretical principles of digital-to-analog and analog-to-digital conversion.

Limitations:

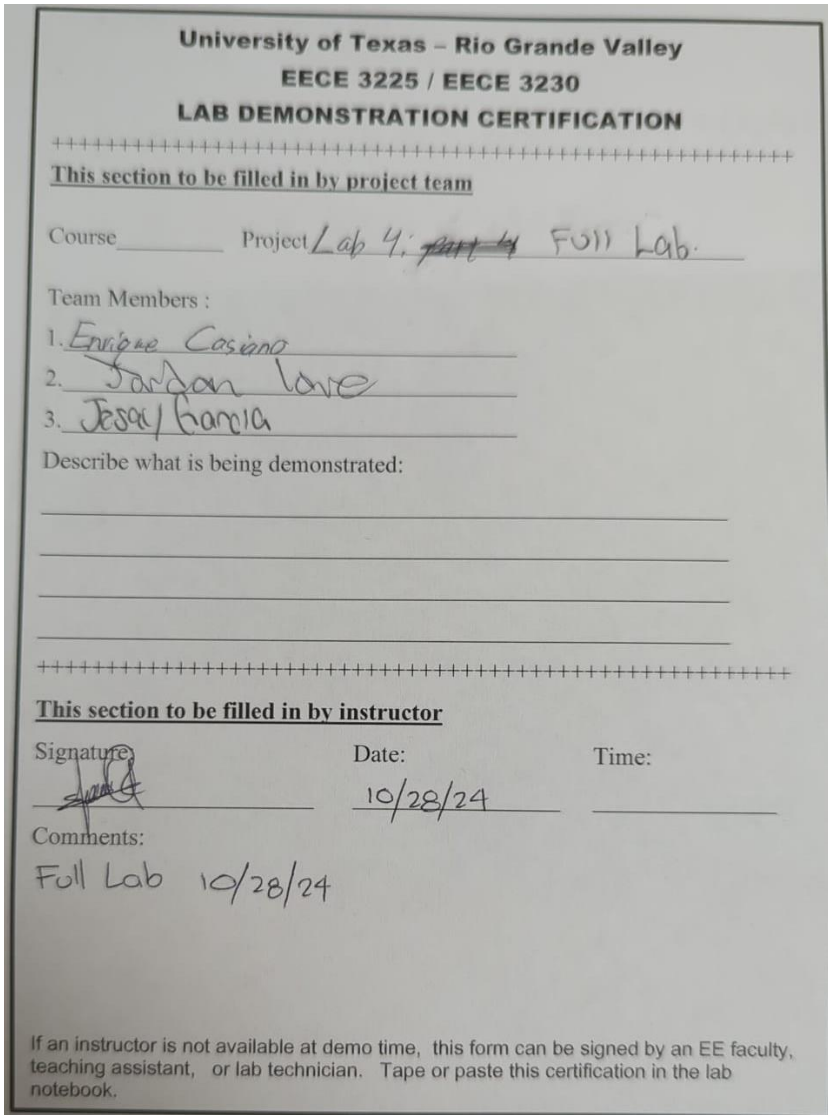
Some limitations emerged due to resistor tolerances in the R/2R ladder network, which caused slight deviations from the expected output voltage values. Additionally, the microcontroller’s processing speed limited the waveform’s resolution and smoothness, with minor delays evident in the stepped waveform approximation. ADC conversions were subject to minor offset errors at lower voltage levels, likely from small variances in the reference voltage and inherent limitations in the microcontroller’s ADC configuration.

Suggestions for Future Research:

Future experiments could explore higher-resolution DAC and ADC implementations, such as a 10-bit or 12-bit DAC, to increase waveform smoothness and accuracy. Testing the DAC with more complex waveform shapes, such as triangular or sawtooth waves, could further demonstrate the microcontroller’s versatility in analog signal generation. Additionally, future research could focus on reducing timing delays through improved programming techniques or utilizing higher-speed microcontrollers, enhancing the DAC’s performance in real-time signal processing applications.

In conclusion, the experiment’s objectives were met, with successful DAC waveform synthesis and ADC functionality demonstrating a clear connection between digital input values and corresponding analog outputs. This hands-on experience with DAC and ADC circuits provided valuable insights into the practical application of digital-to-analog and analog-to-digital conversion theories.

Signatures



University of Texas – Rio Grande Valley
EECE 3225 / EECE 3230
LAB DEMONSTRATION CERTIFICATION

++

This section to be filled in by project team

Course________ Project Lab 4: ~~part 4~~ Full Lab.

Team Members :

1. Enrique Casiano
2. Jordan Love
3. Jesal Garcia

Describe what is being demonstrated:

++

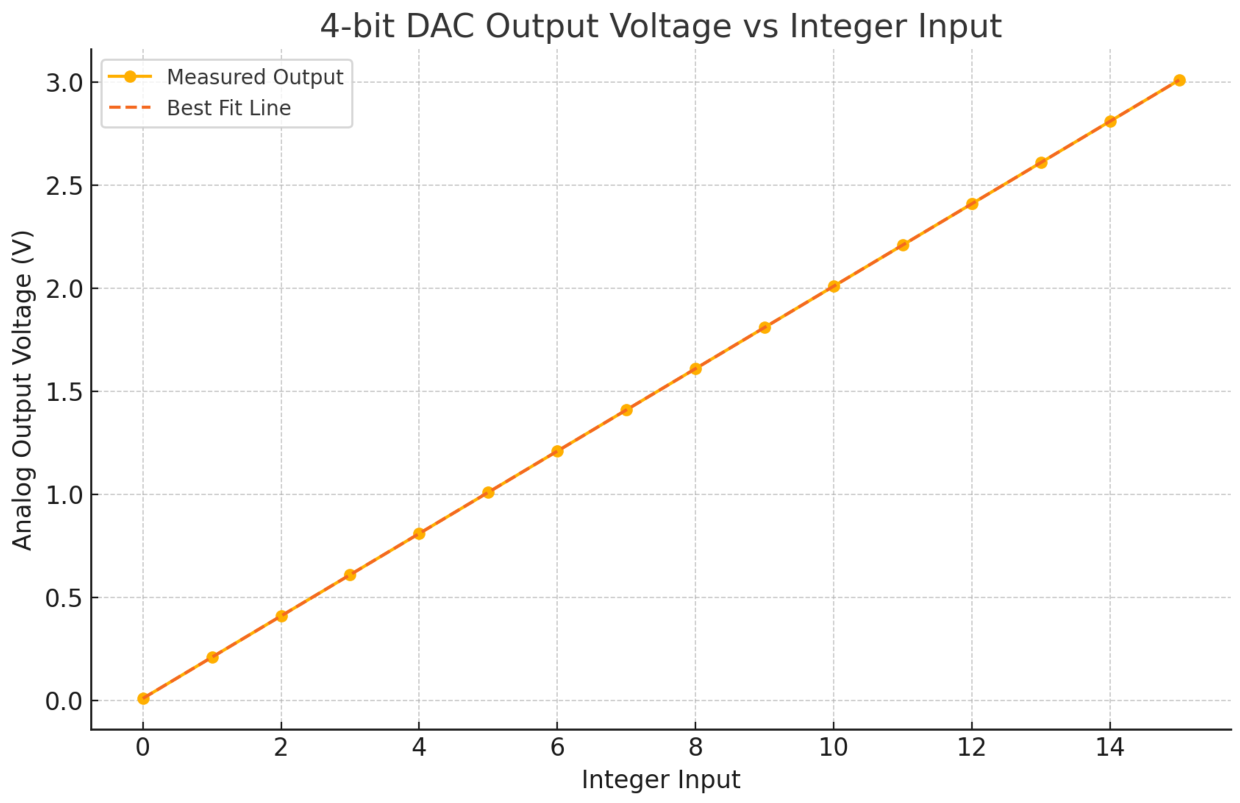
This section to be filled in by instructor

Signature: [illegible] Date: 10/28/24 Time:

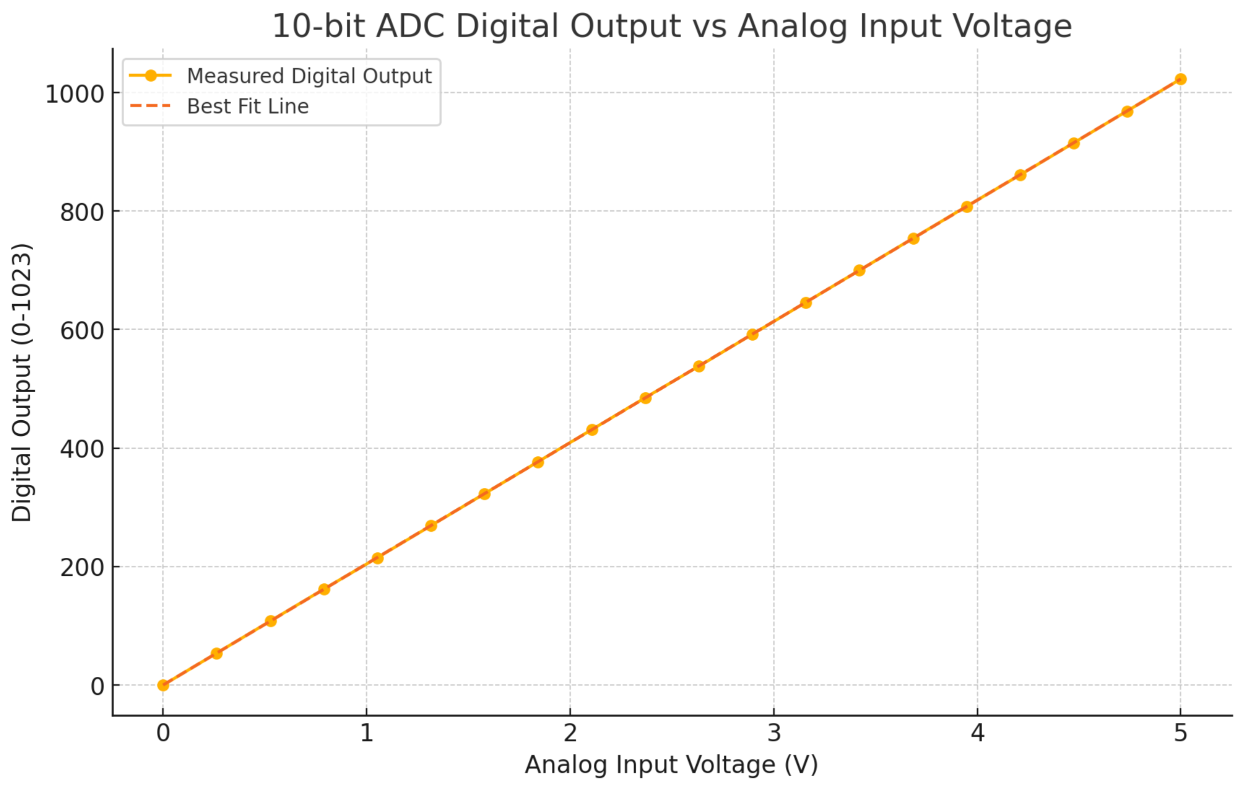
Comments:
Full Lab 10/28/24

If an instructor is not available at demo time, this form can be signed by an EE faculty, teaching assistant, or lab technician. Tape or paste this certification in the lab notebook.

After the Lab



- **Offset Voltage:** Approximately 0 V (no significant offset).
  Offset Voltage=Measured Output at Input 0−Predicted Output at Input 0
- **Full Scale Voltage:** 3.01 V (measured at the maximum integer input).
  Full Scale Voltage=Measured Output at Maximum Integer Input
- **Linearity (Maximum Deviation from Best Fit Line):** Approximately 8.88×10^{-16} V, indicating an almost perfect linear fit with minimal deviation.
  Linearity=max(∣Measured Output−Predicted Output∣)



- **Offset Voltage:** Approximately −0.13 V, indicating a slight offset at zero input. Offset Voltage=Digital Output at 0V−Predicted Output at 0V
- **Full Scale Voltage:** 1023 (reached at the maximum input voltage of 5V). Full Scale Voltage=Digital Output at Maximum Input Voltage (5V)
- **Linearity (Maximum Deviation from Best Fit Line):** Approximately 0.44, suggesting a minimal deviation from ideal linearity. Linearity=max(∣Digital Output−Predicted Output∣)