



Homework 3: Microprocessors

Student: Sergio Andrés Becerra Gándara 0244963

Professor: Dr. Luis Emilio Tonix Gleason

September 15, 2025

1 Introduction

In the previous assignment (HWK2), a polynomial regression function was implemented to map readings from an analog-to-digital converter (ADC) to calibrated values. The objective of this assignment is to compare the computational performance of two methods for performing this mapping on an ESP32 microcontroller:

1. **Direct Calculation:** Evaluating the regression function at runtime.
2. **Look-Up Table (LUT):** Pre-calculating all possible results and storing them in an array for fast and efficient access.

The execution time of both approaches will be measured and analyzed to determine the advantages and disadvantages of each within the context of embedded systems.

2 Look-Up Table (LUT) Generation

To create the LUT, a Python script using the `sympy` library was employed. This script symbolically defines the regression function and evaluates it for each of the 4096 possible values of a 12-bit ADC (range 0-4095). The results were then stored as integers in a C header file (`lookuptable.h`).

The second-order polynomial regression function used was:

$$f(x) = (4.28297943 \times 10^{-6})x^2 - (3.83561791 \times 10^{-2})x + 93.92003116$$

```
1 import sympy as sp
2
3 # 1. Define the symbolic variable and the polynomial function.
4 x = sp.symbols('x')
5 poly = 4.28297943e-06*x**2 - 3.83561791e-02*x + 93.92003116191847
6
7 # 2. Set the table size for a 12-bit ADC.
8 lut_size = 4096
9
10 # 3. Calculate each table value and convert it to an integer.
11 lookup = [int(poly.subs(x, i)) for i in range(lut_size)]
12
13 # 4. Write the table to the 'lookuptable.h' header file.
14 with open("lookuptable.h", "w") as f:
15     # Add header guards to prevent multiple inclusion errors.
16     f.write("#ifndef LOOKUPTABLE_H\n")
17     f.write("#define LOOKUPTABLE_H\n\n")
18
19     # Define the LUT size as a constant.
20     f.write(f"#define LUT_SIZE {lut_size}\n\n")
21
22     # Declare the array as static and constant.
23     f.write(f"static const int lookup_table[LUT_SIZE] = {{\n")
24
25     # Write the table values with formatting.
26     for i, val in enumerate(lookup):
27         f.write(f"    {val},")
28         if (i + 1) % 8 == 0:
29             f.write("\n")
30
31     f.write("};\n\n")
32
33     # Close the header guard.
34     f.write("#endif // LOOKUPTABLE_H\n")
```

Listing 1: Script `lookuptable.py` for generating the LUT.

3 Implementation and Measurement on ESP32

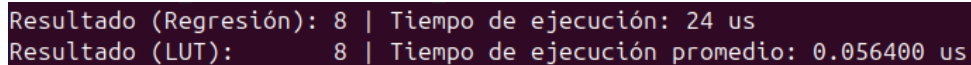
A C program was implemented for the ESP32 to measure and compare the execution times. The `esp_timer_get_time()` function was used to obtain high-precision measurements in microseconds. To accurately measure the LUT's extremely fast look-up time, the access operation was executed 10,000 times in a loop, and the average time was calculated.

```
1 #include <stdio.h>
2 #include "esp_timer.h"
3 #include "lookuptable.h"
4
5 int regression_func(int x) {
6     return (int)(4.28297943e-06*x*x - 3.83561791e-02*x + 93.92003116);
7 }
8
9 void app_main(void) {
10     int test_val = 1234;
11     int result;
12     int64_t start_time, end_time;
13
14     // --- Time Measurement: Regression Method ---
15     start_time = esp_timer_get_time();
16     result = regression_func(test_val);
17     end_time = esp_timer_get_time();
18     printf("Result (Regression): %d | Execution Time: %lld us\n", result, (end_time -
19 start_time));
20
21     // --- Time Measurement: LUT Method with Loop ---
22     const int NUM_ITERATIONS = 10000;
23
24     start_time = esp_timer_get_time();
25     for (int i = 0; i < NUM_ITERATIONS; i++) {
26         result = lookup_table[test_val];
27     }
28     end_time = esp_timer_get_time();
29
30     int64_t total_time_lut = end_time - start_time;
31     double average_time_lut = (double)total_time_lut / NUM_ITERATIONS;
32
33     printf("Result (LUT): %d | Average Time: %f us\n", result, average_time_lut);
34 }
```

Listing 2: Code `main.c` for measuring and comparing execution times.

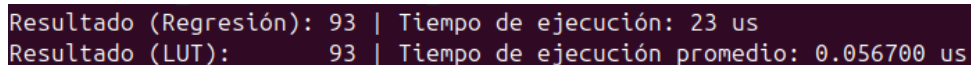
3.1 Results

Running the code 3 times with different values on the ESP32 yielded the times shown in the images below. The regression method required multiple floating-point operations, resulting in a measurable execution time. On the other hand, the LUT look-up, which consists of a single memory access, was so fast that an average over multiple iterations was needed to quantify its duration.



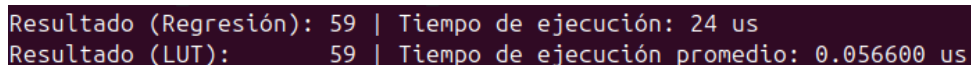
```
Resultado (Regresión): 8 | Tiempo de ejecución: 24 us
Resultado (LUT):      8 | Tiempo de ejecución promedio: 0.056400 us
```

Figure 1: Results with ADC Raw Value = 4095



```
Resultado (Regresión): 93 | Tiempo de ejecución: 23 us
Resultado (LUT):      93 | Tiempo de ejecución promedio: 0.056700 us
```

Figure 2: Results with ADC Raw Value = 0



```
Resultado (Regresión): 59 | Tiempo de ejecución: 24 us
Resultado (LUT):      59 | Tiempo de ejecución promedio: 0.056600 us
```

Figure 3: Results with ADC Raw Value = 1000

As we can observe, the LUT works very well, giving a fast response with the correct values. We can see the results at the left side of the "|" symbol, which represents corresponding humidity % value for the corresponding Raw ADC Value. If we calculate an average of the execution time we obtain this table:

Calculation Method	Execution Time
Regression Calculation	23.6666 μ s
LUT Look-up (average)	0.05656 μ s

Table 1: Comparison of execution times on the ESP32.

4 Conclusion

The results conclusively demonstrate that the LUT is approximately 407 times faster than the direct regression calculation (this value was obtained by dividing 23 μ s and 0.0566 μ s). While the regression takes 23 μ s, the LUT performs the same task in approximately 56 nanoseconds. The LUT offers outstanding speed at the cost of storage memory (in this case, 16 KB for 4096 integers). The direct calculation consumes no significant additional memory but is considerably slower. In conclusion, for applications where latency is critical and sufficient memory is available, the LUT is the preferred optimization strategy. For systems with severe memory constraints, real-time calculation is the only viable alternative, assuming the higher latency is acceptable.