Lab 3: Floating Point vs. Fixed Point

Department of Electrical and Computer Engineering
ECE342, Winter 2026
University of Toronto

Introduction

This lab is designed to deepen your understanding of performance measurement and signal generation in embedded systems. You will explore three important topics:

1. **Timer peripherals:** Learn how to measure the execution time of different code segments with precision.
2. **Fixed-point vs. floating-point operations:** Compare their performance and understand when each is most effective in embedded applications.
3. **Digital-to-analog conversion (in-lab):** Use the DAC on the Nucleo board to generate analog signals from digital data.

Together, these exercises highlight the trade-offs between accuracy, efficiency, and hardware capabilities. By the end of the lab, you will be able to evaluate code performance using timers, make informed choices between fixed- and floating-point arithmetic, and apply DAC output for real-world signal interfacing.

1. Measuring Execution Time

The timer peripheral is a dedicated hardware counter that can be used to measure how many clock cycles an operation requires. It is configured in a similar way to other peripherals you have used. To enable the timer, call `HAL_TIM_Base_Start()`, and to read its current value, use `__HAL_TIM_GET_COUNTER()`.

Measuring execution time: To determine how long a section of code takes to run:

1. Record the timer value before the code segment begins.
2. Record the timer value again after the code segment finishes.
3. Subtract the two values to obtain the number of cycles consumed.

By default, the timer runs on the same clock as the processor and increments once per clock cycle.

Prescaler: Because the timer uses a 16-bit register, it can only count up to 65,536 cycles before rolling over. To extend the measurable range, the timer includes a *prescaler*, which divides the input clock frequency by `prescaler + 1`. For example, if the CPU clock is 100 MHz and the prescaler is set to 3, the timer input clock becomes:

$$\frac{100\,\text{MHz}}{3+1} = 25\,\text{MHz}.$$

By default, the prescaler is 0, meaning the clock is undivided. You will typically leave the prescaler at 0, but increase it when measuring longer operations that exceed 65,536 cycles.

Try it yourself: Use the timer to measure the execution time of a simple code segment, such as a `for` loop that sums the numbers from 0 to 99. How many cycles does this take? Vary the loop bounds and check whether the execution time increases linearly with the loop size.

2. Producing a Sine Wave and Square Wave Using Floating-Point

Waveform generation is a core technique in embedded systems, used in applications such as audio synthesis, communications, and signal testing. In this part of the lab, you will first generate a sine wave using floating-point arithmetic, then extend the method to approximate a square wave using Fourier series.

2.1 Generating a Sine Wave

1. Use the `sin()` function from the `<math.h>` library, which accepts inputs in radians.
2. Inside your `while(1)` loop, iterate over one full period of the sine function in increments of 0.01 radians.
3. For testing, print the generated values over UART.

   Note: Add a small delay (e.g., 1 ms) after each print to avoid overflowing the serial port. This delay would also help with achieving a smooth Sine wave on the oscilloscope. You don't need to present this step, but can be a good debugging point.
4. Once verified, use the timer to measure the time required to compute one full period. **Note:** Comment out any print statements or delays before timing, as they will affect results.

2.2 Hardware vs. Software Floating-Point Computations

The Nucleo-F446ZE board includes a Floating-Point Unit (FPU), which performs floating-point operations efficiently in hardware. The timing you measured above reflects hardware execution. Many lower-cost embedded systems lack an FPU, requiring floating-point operations to be emulated in software, which is significantly slower.
To compare hardware and software execution, recompile your project with the FPU disabled:

Keil IDE:

1. Click `Project > Clean targets`.
2. Open `Project > Options for target lab03`.
3. In the `Target` tab, change `Floating point hardware` from `Single precision` to `Not used`.
4. Recompile the project.

STM32CubeIDE:

1. Right-click the project and select `Properties`.
2. In the left panel, select `C/C++ Build > Settings`.
3. In the right panel, open `MCU Settings`.
4. Change `Floating-point ABI` from `Hardware implementation` to `Software implementation`.
5. Rebuild the project.

Try it yourself: Measure the time required to generate one full period of the sine wave using hardware vs. software floating-point operations.

2.3 Optimizing with a Look-up Table (LUT)

Calling `sin()` repeatedly is computationally expensive, since the same values are recomputed each period. An optimization is to pre-compute sine values once and store them in a *look-up table (LUT)*. During waveform generation, values can then be read directly from the array instead of calling `sin()` each time.

1. Use a `for` loop to compute 314 points over one sine period.
2. Store these values in an array (this loop runs only once).
3. For each period, read values directly from the array instead of calling `sin()`.

This optimization will be useful later when implementing fixed-point arithmetic, where no built-in `sin()` function exists.

2.4 Generating Arbitrary Waveforms

Sine waves can be combined to produce other waveforms, including square, sawtooth, and triangular waves. In this lab, you will focus on the square wave, described by the Fourier series:

$$f(x) = 0.5 + \frac{2}{\pi} \sum_{n=0}^{\infty} \frac{1}{2n+1} \sin\big((2n+1)x\big)$$

This series shows that a square wave can be approximated by summing odd harmonics of sine waves. Using the `sin()` function (or your LUT), write code to approximate a square wave by summing multiple terms. Since you cannot add infinitely many terms, vary the number of terms (n) and observe the effect:

- Execution time increases as more terms are added.
- The waveform shape becomes closer to an ideal square wave with more terms.

Deliverable: Record execution times for $n = 1$ to 6 in the results table provided at the end of this document.

3. Producing a Square Wave Using Fixed-Point Arithmetic

Fixed-point arithmetic is widely used in embedded systems where floating-point hardware is unavailable or too costly. It relies only on integer hardware, making it faster and more power-efficient on low-cost devices. In this part of the lab, you will reproduce the square wave you generated with floating-point arithmetic, but now using fixed-point operations. Unlike floating-point, fixed-point does not provide a built-in `sin()` function, so the LUT approach introduced earlier becomes necessary.

Using a Single LUT for the Square Wave: You already have a LUT for `sin(x)`. For the square wave, you also need terms like `sin(3x)`, `sin(5x)`, etc. These can be derived from the existing LUT by stepping through it at different rates (e.g., `sin(3x)` corresponds to advancing three times faster through the table). Experiment with a few values of x using a calculator to confirm this relationship.

Using Fixed-Point Operations: You are provided with a `fixedpt.h` file, similar to the `fixedpt.c` file from Assignment 2. Unlike the `.c` version, which used regular functions, the `.h` version uses `#define` macros. Macros are expanded by the preprocessor before compilation, avoiding function call overhead and improving performance.
By default, the provided code uses a $Q3.28$ fixed-point format. Proceed as follows:

1. Compute sine values for one full period using `sin()` in floating-point.
2. Convert these values to fixed-point and store them in an array.
3. Write code to generate the square wave using fixed-point operations.
4. Measure execution time as you vary the number of terms added in the Fourier series.

4. Outputting Values Using the DAC (In-Lab)

In the final portion of this lab, you will bridge computation with hardware by outputting your square waves through the DAC and observing them on an oscilloscope. This step connects the numerical waveforms you have generated to real-world signals. The Nucleo board provides a 12-bit DAC, which accepts values from 0 to 4095. Since your computed waveforms do not naturally span this range, you must scale and offset the values so they fit within the DAC's limits. Perform this adjustment for both the floating-point and fixed-point versions of your code.
It is recommended that you first experiment by printing values (e.g., on your laptop) to confirm that the waveform spans the correct range. Once verified, copy your code to the Nucleo board and add the instructions to output the calculated values via the DAC. The DAC output is available on **GPIO `PA_4` (header CN_7)**.
After your code is running on the Nucleo board, make the following modifications to streamline the in-lab demo:

1. Demonstrate a square wave using **floating-point arithmetic**, varying the number of terms (n) from 1 to 6.
2. Demonstrate a square wave using **fixed-point arithmetic**, varying the number of terms (n) from 1 to 6.

Configure the `B1` push-button on the board to cycle through each stage. This allows you to demo all 12 cases without reprogramming the board. **Note:** Comment out any print statements during the demo, as they will affect execution timing.

Demo Checklist:

- Scale values to fit the DAC range (0–4095).
- Verify output by printing values before using the DAC.
- Connect oscilloscope to **PA_4 (CN_7)**.
- Configure `B1` push-button to cycle through 12 stages.
- Comment out all print statements before demo.

5. In-Lab Demo

During the lab session, you must:

- Present the completed results page to your TA.
- Demonstrate your code outputting a square wave via the DAC using both floating-point and $Q3.28$ (same as $Q4.28$) fixed-point formats.

As recommended above, configure the push-button to switch quickly between formats and term counts to keep the demo efficient. You must present the filled tables below during evaluation.

RESULTS

Generating Sine Waves

	Floating-Point		Fixed-Point (Q3.28, same as Q4.28)
	sin()	LUT	LUT
Execution cycles			

Generating Square Waves

Execution time (in cycles)

Number of terms	1	2	3	4	5	6
Floating-Point						
Q3.28 (same as *Q*4.28) Fixed-Point						