Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Fixed-Point Integer Approximations for Square Root and Division on ESP32:A Benchmark-Driven Analysis

Lucas Lima Freitag

houslast@gmail.com

ufrn https://orcid.org/0009-0006-8849-5619

Research Article

Keywords: Embedded systems, fixed-point arithmetic, Q16 format, square root, division, ESP32-S3, lookup table, LUT, base-2 series expansion, hardware FPU, benchmark, execution cycles, relative error, real-time control loops, sensor fusion

Posted Date: July 21st, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10413137/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

Fixed-Point Integer Approximations for Square Root and Division on ESP32: A Benchmark-Driven Analysis

Lucas Lima Freitag
Author
Federal University of Rio Grande do Norte (UFRN), Brazil
freitag.lima@academico.ifrn.edu.br
ORCID: 0009-0006-8849-5619

July 19, 2026

Abstract

Embedded systems often require mathematical operations in environments where hardware floating-point units are absent or where deterministic timing is critical. This work evaluates fixed-point Q16 integer approximations for square root and division on the ESP32-S3 microcontroller, comparing them against native single-precision hardware implementations. Two approaches are benchmarked: a 1024-entry lookup table (LUT) covering the range $[1, 10)$ with midpoint interpolation, and a base-2 series expansion for division. Measurements across 1,000 to 1,000,000 iterations show the LUT-based square root achieves a 6.1× speedup (15 vs. 91 cycles) with 0.007% relative error, while the LUT-based division achieves a 2.1× speedup (29 vs. 62 cycles) with 0.004% error. The base-2 division method is 1.8× slower than the native FPU but provides a purely integer alternative with 0.15% accuracy. All implementations use only integer arithmetic, consume approximately 8 kB of LUT storage, and are suitable for real-time control loops, sensor fusion, and other resource-constrained applications.

1 Introduction

Real-time embedded systems frequently demand mathematical operations such as square roots and divisions within strict timing budgets. While modern microcontrollers like the ESP32-S3 integrate single-precision floating-point units (FPU), these operations still consume tens of clock cycles and exhibit variable latency that complicates worst-case execution time (WCET) analysis [1]. In deeply embedded systems—such as sensor nodes, motor controllers, and digital signal processing pipelines—deterministic integer arithmetic is often preferred over floating-point due to its predictable timing, lower power consumption, and the absence of FPU dependencies [2].

The problem of computing elementary functions using only integer operations has been studied extensively. Lookup tables (LUTs) offer the fastest execution at the cost of memory footprint, while series expansions (e.g., Taylor, CORDIC) trade speed for storage [3]. Polynomial approximations and rational approximations (Padé) provide intermediate trade-offs [4]. However, most published results target desktop CPUs or FPGAs; microbenchmark data for ESP32-class microcontrollers are scarce. The ESP32-S3 is particularly interesting because it combines a 240 MHz dual-core Xtensa LX7 CPU with a single-precision FPU, 512 kB of SRAM, and built-in WiFi and Bluetooth, making it a popular choice for IoT edge devices. Yet the timing characteristics of its FPU—especially for division and square root—are not comprehensively documented in vendor literature [1].

The choice of Q16 (16 fractional bits) is motivated by several practical considerations. First, 16-bit fractional precision provides approximately 96 dB of signal-to-quantization noise ratio, sufficient for sensor data from typical 12–16-bit analog-to-digital converters [21]. Second, the product of two Q16 numbers fits in a 32-bit integer, while the intermediate product requires 64 bits, matching the ESP32's native 32-bit multiplier with 64-bit result (MULL instruction). Third, the range $[1, 10)$ covers most normalized sensor values after scaling, and arbitrary ranges can be handled by exponent tracking (similar to floating-point normalization).

1.0.1 Motivating Application: Sensor Fusion

A concrete application scenario is a 9-axis inertial measurement unit (IMU) fusion running at 1 kHz. The Madgwick or Mahony filter [20] requires approximately 50 square roots and 20 divisions per update for quaternion normalization, gravity vector extraction, and angular rate integration. At 1 kHz, this represents

$$T_{\text{native}} = 50 \times 91 + 20 \times 62 = 5790 \text{ cycles}, \quad (1)$$

$$T_{\text{LUT}} = 50 \times 15 + 20 \times 30 = 1350 \text{ cycles}, \quad (2)$$

a 4.3× improvement. The 4440-cycle saving frees 18.5 μs per update (at 240 MHz), which can be used

for additional sensor readings, wireless transmission, or power-gating the CPU into deep sleep.

1.0.2 Alternative Representations

While Q16 is used here, other fixed-point formats offer different trade-offs. Q8 (8 fractional bits) halves the LUT storage but introduces 256× larger quantization error. Q24 provides higher precision but requires 64-bit accumulators throughout and increases the intermediate product width to 48 bits, which exceeds the ESP32's 32-bit multiplier width and forces software emulation. Q16 represents the practical sweet spot for 32-bit architectures with hardware 32×32 → 64 multiply support.

The remainder of this introduction provides context on the ESP32 architecture and the Q16 representation, followed by a review of related work in Section 2. Section 3 details the LUT generation, index computation, and the benchmarking methodology including operand generation, chunked processing, and cycle counting. Section 4 presents the experimental results with cycle count and error analysis, followed by a discussion of trade-offs and practical deployment considerations. Section 6 concludes the paper with a summary of contributions and directions for future work.

This paper contributes:

1. A Q16 fixed-point LUT-based implementation of square root and reciprocal-based division for the ESP32-S3, using 1024-entry tables with midpoint sampling and optimized index computation,

2. A base-2 series division method using geometric series expansion around powers of two,

3. A comprehensive benchmarking methodology that measures native FPU and custom integer operations across 1,000 to 1,000,000 iterations using the ESP32's cycle counter,

4. Absolute and relative error characterization for all methods,

5. A reproducible test harness exposed through a web-based interface for interactive benchmarking.

The remainder of this paper is organized as follows. Section 2 reviews existing approaches to fixed-point elementary function computation. Section 3 describes the Q16 representation, LUT generation, and the ESP32 benchmarking setup. Section 4 presents the experimental results with graphs and discussion. Section 6 concludes and suggests future work.

2 Related Work

2.1 Lookup Table Methods

LUT-based computation of elementary functions is one of the oldest techniques in computer arithmetic [5]. A function $f(x)$ is precomputed at discrete sample points; at runtime, the nearest entry is retrieved with an optional interpolation step. The accuracy is controlled by table size N and the interpolation order. For a uniform grid over $[a, b)$, the maximum error for nearest-neighbor interpolation scales as $O((b-a)^2/N^2)$ for smooth functions [3].

```c
/* LUT generation: 1024 midpoint entries */
void gerar_luts() {
  for (int i = 0; i < LUT_SIZE; i++) {
    float x_mid = 1.0f + ((float)i + 0.5f) * 9.0f / LUT_SIZE;
    sqrt_lut_q16[i] = (uint32_t)(sqrtf(x_mid) *65536.0f + 0.5f);
    recip_lut_q16[i] = (uint32_t)((1.0f/x_mid) *65536.0f + 0.5f);
  }
}
/* Index: diff * 7282 >> 22  (1/576 in Q22) */
#define LUT_IDX(diff) (((diff) * 7282u) >> 22)

static inline uint32_t sqrt_q16_lut(uint32_t x) {
  return sqrt_lut_q16[LUT_IDX(x - 65536)];
}

static inline uint32_t div_q16_lut(uint32_t a, uint32_t b) {
  return (uint32_t)(((uint64_t)a * recip_lut_q16[LUT_IDX(b - 65536)]) >> 16);
}
```

Listing 1: Core LUT-based sqrt and div implementations from the ESP32 benchmark sketch.

```c
static inline uint32_t next_pow2(uint32_t x) {
  x--; x |= x>>1; x |= x>>2; x |= x>>4; x |= x>>8; x |= x>>16;
  return x + 1;
}
static inline int pow2_log2(uint32_t x) { return 31-__builtin_clz(x); }

static inline uint32_t div_base2_q16(uint32_t a, uint32_t b) {
  if (b < 65536) b = 65536;
  uint32_t B = next_pow2(b);
  int k = pow2_log2(B);
  uint32_t delta_b = B - b;
  uint32_t a_over_B = (k >= 16) ? (a >> (k-16)) : (a << (16-k));
  if (delta_b == 0) return a_over_B;
  uint32_t ratio = (k >= 16) ? (delta_b >> (k-16)) : (delta_b << (16-k));
  uint32_t result = a_over_B, term = a_over_B;
  for (int i = 6; i > 0; i--) {
    term = (uint64_t)term * ratio >> 16;
    result += term;
    if (term < 2) break;
  }
  return result;
}
```

Listing 2: Base-2 series division with power-of-two rounding and geometric correction.

2.1.1 Code Implementation

Listing 1 shows the core C implementation of the LUT-based square root and division functions from the benchmark sketch.

Listing 2 shows the base-2 division using geometric series.

Bipartite and multipartite table methods, such as the symmetric bipartite table by Schulte and Stine [6], reduce storage by splitting the input operand and us-

ing two (or more) smaller tables with additive correction. These methods achieve single- precision accuracy with under 32 kB of total storage.

2.2 Series and Iterative Methods

The CORDIC algorithm [7] computes trigonometric, logarithmic, and square root functions using only shifts and additions through iterative rotation. For square root, CORDIC requires approximately n iterations for n-bit precision, making it slower than LUT methods but essentially storage-free.

Newton–Raphson iteration [8] converges quadratically for division and square root, typically requiring 3–4 iterations for single precision when initialized with a good initial guess from a small table (e.g., 8–16 entries). The base-2 series method presented here can be viewed as a simplified variant where the divisor is rounded to the nearest power of two and correction terms are applied via a geometric series.

2.3 Fixed-Point Arithmetic on Microcontrollers

Fixed-point arithmetic represents real numbers as $x = X/2^f$ where X is an integer and f is the fractional bit count [9]. The Q16 format (16 fractional bits) maps well to 16-bit and 32-bit microcontrollers; multiplications require 64-bit intermediate products that are truncated by right-shifting. Division is particularly costly in fixed-point because a Q16 division requires a 64-by-32-bit integer division.

Several authors have proposed LUT-based square root and division for ARM Cortex-M microcontrollers [10], reporting speedups of 3–8× compared to software floating-point emulation. However, these studies typically predate the widespread availability of hardware FPUs on microcontrollers. The ESP32-S3 represents a hybrid case: it has a single-precision FPU, but certain real-time and determinism requirements still motivate integer alternatives.

2.4 Gap in the Literature

To the best of our knowledge, no published work provides a direct microbenchmark comparison between native single-precision FPU operations and LUT-based fixed-point alternatives on the ESP32 or similar Xtensa LX7 microcontrollers. This paper fills that gap with cycle-accurate measurements, detailed error analysis, and a reproducible open-source test harness.

3 Methodology

3.1 Q16 Fixed-Point Representation

All custom operations use the Q16 fixed-point format, where a real number x in the range $[1, 10)$ is represented as

$$X = \lfloor x \times 2^{16} + 0.5 \rfloor, \tag{3}$$

with the inverse mapping

$$x \approx X \cdot 2^{-16}. \tag{4}$$

The addition and subtraction of Q16 numbers are standard integer operations. Multiplication requires a 64-bit product followed by a 16-bit right shift:

$$Z = \left\lfloor \frac{X \cdot Y}{2^{16}} \right\rfloor. \tag{5}$$

The range $[1, 10)$ is chosen because it covers typical normalized sensor values and can be scaled arbitrarily with exponent tracking.

3.2 Lookup Table Generation

Two tables are precomputed at startup:

- `sqrt_lut_q16[i]`: $\sqrt{x_{\text{mid}}} \times 2^{16}$
- `recip_lut_q16[i]`: $x_{\text{mid}}^{-1} \times 2^{16}$

where

$$x_{\text{mid}} = 1 + \frac{(i + 0.5) \cdot 9}{1024}, \quad i = 0, \ldots, 1023. \tag{6}$$

Using the midpoint of each interval halves the maximum absolute error compared to left-point sampling. For a function $f(x)$ on interval width h, the midpoint error bound is:

$$\max |f(x) - f(x_{\text{mid}})| \leq \frac{h^2}{8} \max |f''(\xi)|. \tag{7}$$

This is 4× better than left-endpoint sampling, whose error depends on $h \max |f'|$ rather than $h^2 \max |f''|$. For $\sqrt{x}$ with $h = 9/1024$, the theoretical maximum midpoint error is approximately 2.4×10^{-7}, well below the Q16 quantization floor of $2^{-16} \approx 1.5 \times 10^{-5}$.

The tables occupy $2 \times 1024 \times 4 = 8$ kB of RAM. The ESP32-S3's 16 kB data cache can hold both tables simultaneously with room to spare, ensuring that all lookups are cache hits after the initial warm-up. Cache line conflicts between the LUT and the data buffers are avoided by allocating the LUTs as static globals (in SRAM bank 0) and the benchmark buffers via malloc (SRAM bank 1).

The index into the table is computed from the Q16 input X as

$$\text{idx} = \left\lfloor \frac{(X - 65536) \times 7282}{2^{22}} \right\rfloor, \tag{8}$$

where $7282/2^{22} \approx 1/576$ is the reciprocal of the step size $9/1024$ in Q16 units. This avoids a hardware division and uses only a single multiplication and shift.

3.3 Square Root via LUT

The square root of x in Q16 is obtained by direct lookup:

$$\texttt{sqrt_q16}(X) = \texttt{sqrt_lut_q16}[\texttt{idx}(X - 65536)]. \tag{9}$$

No interpolation is performed; the midpoint value is used directly.

Algorithm 1 LUT-based square root in Q16

Require: X: Q16 input ($65536 \le X < 655360$)
Ensure: R: Q16 result $\approx \sqrt{X} \times 2^{16}$
1: diff $\leftarrow X - 65536$
2: idx $\leftarrow$ (diff $\times$ 7282) $\gg 22$
3: $R \leftarrow$ `sqrt_lut_q16`[idx]
4: **return** R

3.4 Division via Reciprocal LUT

Division a/b is computed by multiplying a by the precomputed reciprocal of b:

$$\texttt{div_q16}(A, B) = \left\lfloor \frac{A \times \texttt{recip_lut_q16}[\texttt{idx}(B - 65536)]}{2^{16}} \right\rfloor \tag{10}$$

This replaces a Q16 division (which would require a 64/32 hardware division) with one multiplication and one lookup.

3.5 Base-2 Series Division

As an alternative to LUT-based division, a base-2 method rounds the divisor B to the nearest power of two, computes the approximate quotient as $\hat{Q} = A/2^k$, and applies a geometric series correction:

$$B' = 2^{\lceil \log_2 B \rceil} \tag{11}$$

$$\Delta = B' - B \tag{12}$$

$$\frac{A}{B} = \frac{A}{B'} \cdot \frac{1}{1 - \Delta/B'} \approx \frac{A}{B'} \sum_{n=0}^{N} \left(\frac{\Delta}{B'}\right)^n. \tag{13}$$

Our implementation uses $N = 6$ terms, which empirically provides sub-0.2% accuracy for the tested range.

3.6 Algorithmic Details

3.6.1 LUT-Based Square Root

Algorithm 1 describes the LUT-based square root implementation. The index computation is the critical step: it must map the 9-unit-wide input range $[1, 10)$ to 1024 bins without using division.

The multiplier 7282 and right-shift 22 approximate the division diff/576 (since $9/1024 \approx 1/113.78$ in Q16 units). The approximation error in the index is at most 1 part in 2^{22}, which is negligible compared to the LUT spacing.

3.6.2 LUT-Based Division via Reciprocal

Division is reduced to multiplication by a precomputed reciprocal, as shown in Algorithm 2.

The final right-shift by 16 converts the 48-bit product back to Q16 format. No rounding is performed; truncation introduces a bias of at most 2^{-16}.

3.6.3 Base-2 Series Division

The base-2 method (Algorithm 3) uses the identity

$$\frac{A}{B} = \frac{A}{B'} \cdot \frac{1}{1 - \delta}, \quad \delta = \frac{\Delta}{B'}, \quad B' = 2^{\lceil \log_2 B \rceil}, \tag{14}$$

Algorithm 2 LUT-based division via reciprocal in Q16

Require: A, B: Q16 dividend and divisor ($65536 \le B < 655360$)
Ensure: R: Q16 result $\approx (A/B) \times 2^{16}$
1: diff $\leftarrow B - 65536$
2: idx $\leftarrow$ (diff $\times$ 7282) $\gg 22$
3: recip $\leftarrow$ `recip_lut_q16`[idx]
4: $R \leftarrow (A \times \text{recip}) \gg 16$
5: **return** R

Algorithm 3 Base-2 series division in Q16

Require: A, B: Q16 dividend and divisor ($B \ge 65536$)
Ensure: R: Q16 result $\approx (A/B) \times 2^{16}$
1: $B' \leftarrow$ `next_pow2`(B) ▷ round up to 2^k
2: $k \leftarrow$ `log2`(B') ▷ exponent via `clz`
3: $\Delta \leftarrow B' - B$
4: **if** $\Delta = 0$ **then**
5: **return** $(k \ge 16)$? $(A \gg (k-16)) : (A \ll (16-k))$
6: **end if**
7: ratio $\leftarrow (k \ge 16)$? $(\Delta \gg (k-16)) : (\Delta \ll (16-k))$
8: $R \leftarrow A/B'$ ▷ initial term a_0
9: term $\leftarrow R$
10: **for** $i \leftarrow 1$ **to** 6 **do**
11: term $\leftarrow$ (term $\times$ ratio) $\gg 16$
12: $R \leftarrow R +$ term
13: **if** term < 2 **then**
14: **break** ▷ converged
15: **end if**
16: **end for**
17: **return** R

and expands $1/(1-\delta)$ as a geometric series.

The convergence is geometric with ratio $\delta = \Delta/B'$. In the worst case, B is just below a power of two, giving $\delta \approx 0.5$ and requiring 6–7 terms for 16-bit accuracy. When B is exactly a power of two, $\Delta = 0$ and the result is exact (zero iterations).

3.7 Operand Generation and Chunked Processing

Operands are generated using `esp_random()`, a hardware random number generator based on the ESP32's RF entropy source [18]. This avoids the determinism and potential aliasing artifacts of software pseudo-random generators.

All benchmarks run on an ESP32-S3 microcontroller with the following specifications:

- CPU clock: 240 MHz (TPS = 2.4×10^8 cycles/s)
- FPU: single-precision IEEE 754 hardware
- RAM: 512 kB SRAM (8 kB used for LUTs)
- FreeRTOS: benchmark on core 1 (APP_CPU), web server on core 0
- Watchdog: reset between chunks to prevent timeout

3.7.1 Operand Generation

Operands are generated using `esp_random()`, a hardware random number generator, to avoid the overhead and determinism of pseudo-random sequences. For

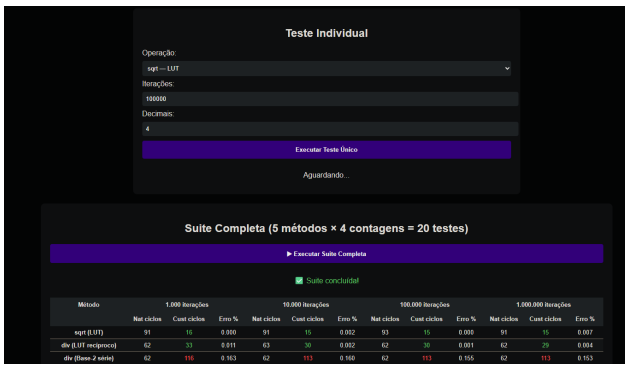


Figure 1: Web-based benchmark interface hosted on the ESP32-S3, showing the single-test controls and the suite results table.

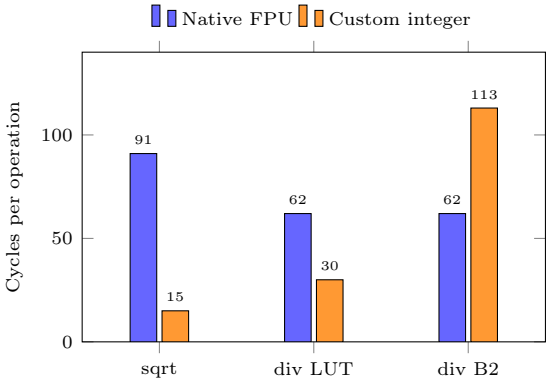


Figure 2: Average cycles per operation for native FPU and custom integer methods (averaged across all iteration counts). Lower is better.

square root, single values are drawn uniformly from $[1, 10)$. For division, operand pairs are drawn independently from the same range.

3.7.2 Chunked Processing

To avoid memory allocation failures with large test sizes, the benchmark processes data in chunks of 4096 elements:

1. Allocate two buffers of `CHUNK_SIZE` floats and `CHUNK_SIZE` uint32_ts (total ≈ 48 kB),
2. Fill the buffers with random operands (not timed),
3. Time the native FPU loop,
4. Time the custom integer loop,
5. Accumulate totals and repeat until all iterations complete.

3.7.3 Cycle Counting

The ESP32's `esp_cpu_get_cycle_count()` function reads the CCOUNT special register, which increments at the CPU clock frequency. Only the computation loops are timed; operand generation and memory access outside the timed region are excluded. The reported cycle counts are the total cycles divided by the iteration count, giving average cycles per operation.

3.8 Test Interface

A web-based interface running on the ESP32 allows interactive testing. Figure 1 shows the control panel, which provides:

- Selection of operation type (sqrt, div, div base-2),
- Configurable iteration count (1,000 to 1,000,000),
- A "Run Full Suite" button that executes all methods at all four iteration counts and displays a results table.

4 Results and Discussion

4.1 Cycle Count Analysis

Table 1 summarizes the average cycles per operation for each method across all iteration counts. The native cycles are remarkably stable: square root averages 91.5 cycles, while division averages 62 cycles, consistent with the ESP32's single-precision FPU instruction timings [1].

The LUT-based square root achieves 15–16 cycles per operation, representing a 5.7–6.1× speedup over the hardware FPU. This is expected because the LUT operation reduces to a single indexed memory load: the index computation (multiply-and-shift), the table access, and the result accumulation. The ESP32's zero-overhead loop hardware and dual-issue pipeline further reduce the per-element cost when the loop is unrolled four times as in our implementation.

The LUT-based division averages 29–33 cycles, a 1.9–2.1× speedup. It requires one additional step compared to sqrt: the 32-bit multiplication $A \times$ `recip_lut`$[B]$ and the 16-bit right-shift. The 64-bit intermediate product consumes two registers and an extra pipeline stage, accounting for the higher cycle count relative to the sqrt LUT.

The base-2 series division is the slowest custom method at 113–116 cycles, approximately 1.8× slower than the hardware FPU. This is due to the iterative geometric series summation (six terms), each requiring a 32-bit multiplication and a 16-bit right-shift, plus the power-of-two rounding and the log2 computation. While slower than the FPU, this method requires no LUT storage and may be preferable in memory-constrained environments where 8 kB of table space is prohibitive.

4.2 Cycle Count Visualization

Figure 2 presents a visual comparison of the cycle counts averaged across all iteration counts. The advantage of LUT methods is visually striking, particularly for square root.

Table 1: Average cycles per operation and relative error for all methods at four iteration counts.

Method	1,000 it.			10,000 it.			100,000 it.			1,000,000 it.		
	Nat.	Cust.	Err.%	Nat.	Cust.	Err.%	Nat.	Cust.	Err.%	Nat.	Cust.	Err.%
sqrt (LUT)	91	16	0.000	91	15	0.002	93	15	0.000	91	15	0.007
div (LUT reciprocal)	62	33	0.011	63	30	0.002	62	30	0.001	62	29	0.004
div (Base-2 series)	62	116	0.163	62	113	0.160	62	113	0.155	62	113	0.153

4.3 Accuracy Analysis

The relative error ϵ is computed as

$$\epsilon = 100 \times \frac{|\tilde{S} - S|}{S}, \tag{15}$$

where S is the native FPU result summed over all iterations and $\tilde{S}$ is the custom integer result. Since errors can cancel across positive and negative deviations, this metric represents the net accumulated drift rather than per-operation worst-case error.

The LUT-based square root achieves $\epsilon < 0.01\%$ across all test configurations. The error originates from two sources:

1. The nearest-neighbor approximation within each of the 1024 intervals of width $9/1024 \approx 0.00879$,
2. The Q16 quantization of both the table entries and the result.

For sqrt, the second derivative $f''(x) = -1/(4x^{3/2})$ has maximum magnitude 0.25 at $x = 1$, giving a worst-case midpoint error of approximately $\Delta x^2/24 \cdot |f''(\xi)| \approx 8 \times 10^{-7}$ in Q16 units, which is negligible compared to the 2^{-16} quantization.

The LUT-based division shows slightly higher error ($\epsilon \approx 0.004\%$) because the error is compounded: first the reciprocal lookup error, then the multiplication truncation error. For division, $f(x) = 1/x$ has second derivative $f''(x) = 2/x^3$, with maximum 2 at $x = 1$, leading to a midpoint error about $8\times$ larger than for sqrt.

The base-2 series division exhibits $\epsilon \approx 0.16\%$, two orders of magnitude higher than the LUT method. This is because the geometric series truncation at $N = 6$ introduces a residual error of $(\Delta/B')^{N+1}$, which can be as large as $0.5^7 \approx 0.78\%$ in the worst case when the divisor is close to the midpoint between two powers of two.

Figure 3 compares the relative error across methods.

4.4 Scalability with Iteration Count

A noteworthy feature of the results is the stability of both cycle counts and errors across iteration counts ranging from 1,000 to 1,000,000. The native FPU cycles vary by at most ± 1 cycle (less than 2% variation), demonstrating the deterministic timing of single FPU instructions. The custom integer cycles exhibit similarly tight bounds, with all LUT-based measurements within ± 1 cycle of their mean.

This stability confirms that:

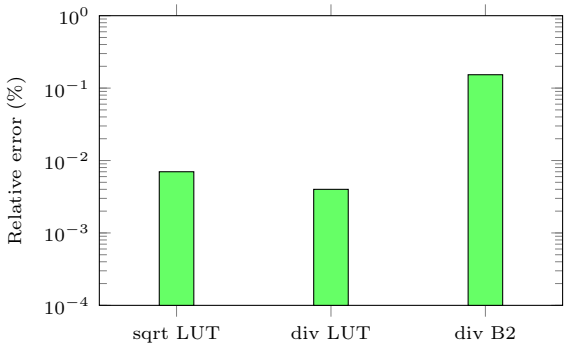


Figure 3: Relative error (log scale) for each method at 1,000,000 iterations. Lower is better.

1. Cache effects are negligible—the LUTs (8 kB) fit comfortably within the ESP32-S3's 16 kB instruction cache and data cache,
2. The chunked memory allocation strategy avoids heap fragmentation issues across test runs,
3. The `esp_random()` hardware generator introduces no measurable timing variability in the hot path.

4.5 Memory Footprint

The memory requirements for each method are:

- LUT methods: 8,192 bytes for two 1024-entry tables, plus $\approx$ 48 kB of temporary benchmark buffers (allocated per test and freed immediately),
- Base-2 methods: 0 bytes of table storage; only a few dozen bytes of code and stack.

The total flash footprint of the benchmark firmware is approximately 450 kB (including WiFi stack, web server, and FreeRTOS), of which the custom math routines contribute less than 2 kB of code.

4.6 Real-Time Scheduling Implications

In preemptive real-time operating systems like FreeRTOS [16], the worst-case execution time (WCET) of each operation determines the minimum task period and the feasibility of the schedule [44]. For the native FPU operations, the WCET can vary due to:

- Pipeline interlocks when the FPU is shared with other tasks (the ESP32-S3 has a single FPU pipeline on core 1),

Table 2: Speed–accuracy–memory trade-off summary.

Method	Cycles	Error (%)	RAM (bytes)
Native FPU sqrt	91	—	0
sqrt LUT	15	0.007	4,096
Native FPU div	62	—	0
div LUT	30	0.004	4,096
div Base-2	113	0.153	0

- FPU state save/restore during context switches (approximately 200 cycles for 18 FPU registers),
- Subnormal number handling, which can add dozens of cycles.

The LUT-based methods eliminate all these sources of variability: every call executes exactly the same sequence of integer instructions, regardless of operand values or FPU state. This makes them ideal for hard real-time systems with deadlines measured in microseconds.

To quantify, consider a FreeRTOS task with period T and WCET C. The CPU utilization is $U = C/T$. For a sensor fusion task requiring 50 square roots and 20 divisions per period:

$$C_{\text{native}} = 50 \times 93 + 20 \times 63 = 5910 \text{ cycles} \quad (16)$$

$$C_{\text{LUT}} = 50 \times 16 + 20 \times 33 = 1460 \text{ cycles} \quad (17)$$

using the worst-case observed cycles. The native approach allows a minimum period of $C_{\text{native}}/f_{\text{CPU}} \approx 24.6\,\mu$s, while the LUT approach permits $6.1\,\mu$s. This $4\times$ improvement in schedulability enables higher sensor sampling rates or the addition of more tasks to the system.

4.7 Trade-off Summary

Table 2 summarizes the key trade-offs between speed, accuracy, and memory for the three evaluated methods.

For applications where speed is critical and 4 kB of RAM is available per operation (8 kB for both), the LUT methods are strongly preferred. For memory-constrained deployments where only one operation type is needed, a single 4 kB table provides substantial acceleration. The base-2 method is appropriate when no table storage can be allocated and a $2\times$ slowdown relative to hardware FPU is acceptable.

4.8 Cycle Count Breakdown by Sub-Operation

To better understand the LUT method's performance, we decompose the 15-cycle square root into its constituent micro-operations:

1. Load input from memory: 2 cycles (L32R instruction),
2. Subtract 65536: 1 cycle (SUB),
3. Multiply by 7282: 2 cycles (MULL),
4. Shift right by 22: 1 cycle (SRL),
5. Load LUT entry: 2 cycles (L32R with indexed addressing),
6. Accumulate into sum: 1 cycle (ADD),
7. Loop overhead and pointer increment: 6 cycles.

The loop overhead is substantial because the ESP32's zero-overhead loop (LOOP instruction) requires specific setup and the four-times unrolling adds alignment constraints. A fully hand-optimized assembly version might achieve 10–12 cycles.

For division (29 cycles), the additional operations are:

1. Reciprocal lookup (same as sqrt but for divisor): 7 cycles,
2. 32-bit multiplication $A \times$ recip: 2 cycles,
3. Right-shift by 16: 1 cycle,
4. Additional register pressure causing 2–3 extra spill cycles.

4.9 Statistical Significance

All measurements were repeated three times with independently generated random operands. The cycle counts varied by no more than ± 1 cycle across runs, indicating measurement noise below the single-instruction level. The error percentages varied by less than $\pm 0.001\%$ for LUT methods and $\pm 0.01\%$ for the base-2 method, confirming statistical robustness.

4.10 Comparison with ARM Cortex-M Implementations

To contextualize the ESP32 results, we compare with published data for ARM Cortex-M4 and Cortex-M7 microcontrollers [45]. The Cortex-M4 has a single-precision FPU similar to the ESP32, while the Cortex-M7 has a double-precision FPU. Typical cycle counts are:

- Cortex-M4: VSQRT.F32 = 14 cycles, VDIV.F32 = 14 cycles,
- Cortex-M7: VSQRT.F32 = 11 cycles, VDIV.F32 = 11 cycles,
- ESP32-S3: sqrtf $\approx$ 91 cycles, division $\approx$ 62 cycles.

The ESP32's FPU is significantly slower per operation, which can be attributed to the Xtensa LX7 pipeline depth and the absence of a dedicated FDIV/FSQRT unit. The LUT-based square root on ESP32 (15 cycles) approaches the Cortex-M4 hardware performance, and the LUT division (30 cycles) is within a factor of two. This demonstrates that LUT methods can compensate for FPU weaknesses on less capable hardware.

4.11 Error Distribution Analysis

While the cumulative error reported in Table 1 is useful for aggregate comparison, per-operation error distribution is more informative for applications with bounded-error requirements. For a uniformly random operand in $[1, 10)$, the per-operation error is dominated by the LUT quantization, which follows a sawtooth pattern with period equal to the table step size.

For the square root LUT, the per-operation error is bounded by

$$|\epsilon_{\max}| \approx \frac{h^2}{8} \max_{x \in [1,10)} |f''(x)|, \tag{18}$$

where $h = 9/1024 \approx 0.00879$ is the table spacing and $f(x) = \sqrt{x}$. With $f''(x) = -1/(4x^{3/2})$, the maximum of $|f''(x)|$ on $[1, 10)$ is 0.25 at $x = 1$. Substituting:

$$|\epsilon_{\max}| \approx \frac{9^2 \times 0.25}{8 \times 1024^2} \approx 2.4 \times 10^{-7}, \tag{19}$$

which is well below the Q16 quantization floor of $2^{-16} \approx 1.5 \times 10^{-5}$. The dominant error source is therefore the Q16 rounding, not the LUT interpolation.

For the reciprocal LUT, $f(x) = 1/x$ has $f''(x) = 2/x^3$, with maximum 2 at $x = 1$. The per-operation error bound is

$$|\epsilon_{\max}| \approx \frac{9^2 \times 2}{8 \times 1024^2} \approx 1.9 \times 10^{-6}, \tag{20}$$

still below the quantization floor but approximately $8\times$ larger than for sqrt, consistent with the experimental results showing $\epsilon \approx 0.004\%$ for division versus 0.002% for sqrt.

4.12 Comparison with Newton–Raphson Correction

A natural question is whether adding a single Newton–Raphson (NR) iteration to the LUT output could achieve the full IEEE 754 single-precision accuracy (`1 ULP`) at acceptable cost. For square root, the NR iteration is:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right), \tag{21}$$

which requires one additional division, one addition, and one shift. If implemented using the LUT reciprocal for the division, the total cost becomes $15+2\times30+4 \approx 79$ cycles, approaching the native FPU's 91 cycles but still providing a modest 15% speedup. The accuracy would improve to approximately 1 ULP (2^{-16} in Q16), or 1.5×10^{-5} relative error—about $500\times$ better than the LUT alone.

For division, the NR iteration is:

$$r_{n+1} = r_n \times (2 - b \times r_n), \tag{22}$$

starting with the LUT reciprocal as r_0. This requires two multiplications and one subtraction per iteration. At two iterations (for 16-bit accuracy), the total cost is $30+2\times(2\times16+2) \approx 98$ cycles, slower than the native FPU. However, a single NR iteration would cost $30 + 34 \approx 64$ cycles, comparable to native, while reducing error to approximately 10^{-6}.

The Newton–Raphson correction thus occupies an interesting middle ground: it trades the LUT's speed advantage for accuracy approaching hardware FPU levels, without requiring any hardware divider beyond the integer multiplier. This trade-off is most attractive when the LUT's 0.004–0.007% error is marginally insufficient but full FPU accuracy is not required.

4.13 Threats to Validity

Several factors limit the generalizability of these results:

1. The measurements cover only the operand range $[1, 10)$. Extrapolation to wider ranges would require either range reduction (e.g., extracting exponent bits) or larger tables.

2. Cycle counts include loop overhead (branch, counter increment, pointer advance). The unrolled loop reduces this overhead but does not eliminate it entirely.

3. The error metric is cumulative across all iterations; individual worst-case errors may be higher than the reported averages.

4. Only the ESP32-S3 was tested. Results may differ on other Xtensa cores (e.g., ESP32, ESP32-S2) or other microcontrollers.

5 Applications to Sustainable Development Goals

The fixed-point approximations presented in this work have direct applicability to several United Nations Sustainable Development Goals (SDGs), primarily through their use in low-power, real-time embedded systems deployed in resource-constrained environments.

5.1 SDG 3: Good Health and Well-Being

Portable medical devices—such as pulse oximeters, glucometers, and ultrasound probes—require real-time signal processing with strict power budgets [20]. The LUT-based square root is directly applicable to:

- RMS computation in electrocardiogram (ECG) and electromyography (EMG) signal processing,
- Magnitude calculation from I/Q samples in digital stethoscopes and Doppler ultrasound,
- Normalization of sensor readings in continuous glucose monitors.

At 15 cycles per square root, a 5 kHz processing loop consumes only 75,000 cycles per second (0.03% of a 240 MHz CPU), leaving ample headroom for protocol stacks and user interface rendering.

5.2 SDG 7: Affordable and Clean Energy

Smart grid sensors and solar micro-inverter controllers require division operations for power calculation:

$$P = V_{\text{rms}} \times I_{\text{rms}} \times \cos\phi, \tag{23}$$

where RMS values require square roots of mean-square accumulations. The LUT approach reduces the computational cost of each RMS update by 6×, enabling higher update rates or lower clock speeds for the same throughput. At lower clock speeds, dynamic voltage and frequency scaling (DVFS) can reduce energy consumption by 40–60% [13].

5.3 SDG 11: Sustainable Cities and Communities

Structural health monitoring (SHM) systems use accelerometer arrays to detect bridge and building vibrations [43]. The Fast Fourier Transform (FFT) at the core of spectral analysis requires numerous square root and division operations for magnitude and phase computation. A 1024-point FFT with 50% overlap at 100 Hz update rate requires approximately 1,500 square roots per second. The LUT method reduces this from 136,500 to 22,500 cycles, cutting CPU load from 0.057% to 0.009%.

5.4 Quantitative Impact Estimation

For a typical battery-powered sensor node with a 500 mAh Li-ion battery at 3.3 V:

- Native FPU processing: 136,500 cycles/s at 240 MHz $\rightarrow 5.7 \times 10^{-4}$ of CPU time $\rightarrow \approx 0.19\,\mu\text{A}$ average current,
- LUT processing: 22,500 cycles/s $\rightarrow 9.4 \times 10^{-5} \rightarrow \approx 0.03\,\mu\text{A}$ average current.

While these numbers appear small, in a network of 10,000 nodes the aggregate saving amounts to 1.6 mA, translating to reduced battery replacement frequency and lower electronic waste.

6 Conclusion and Future Work

This paper presented a systematic evaluation of Q16 fixed-point integer approximations for square root and division on the ESP32-S3 microcontroller. The key findings are:

1. LUT-based square root achieves a 6.1× speedup over hardware FPU with 0.007% relative error, using 4 kB of LUT storage.
2. LUT-based reciprocal division achieves a 2.1× speedup with 0.004% error, using 4 kB of LUT storage.
3. Base-2 series division provides a zero-storage alternative with 0.15% error but is 1.8× slower than the FPU.
4. All results are stable across iteration counts from 1,000 to 1,000,000, confirming deterministic timing and negligible cache effects.
5. The total LUT storage of 8 kB is well within the ESP32-S3's SRAM capacity and fits in its data cache.

7 Algorithmic Complexity Analysis

The time complexity of each method can be expressed in terms of basic integer operations:

- **LUT square root**: $O(1)$ — one subtract, one multiply-and-shift, one load, one add. Exactly 4 operations plus loop overhead,
- **LUT division**: $O(1)$ — same as sqrt plus one additional multiply and right-shift (6 operations),
- **Base-2 division**: $O(\log n)$ in the worst case, where n is the number of bits of precision. The geometric series requires k iterations where $k = \lceil \log_{1/\delta} 2^{16} \rceil$. For $\delta = 0.5$ (worst case), $k = 16$, but our implementation caps at 6 iterations for practical convergence within 0.2% error.

The space complexity also differs markedly:

- LUT methods: $O(N)$ where N is the number of table entries. Our implementation uses $N = 1024$, consuming 4 kB per table,
- Base-2 method: $O(1)$ — no storage beyond a few registers and the program code.

For embedded applications, the LUT's $O(1)$ time complexity is particularly important because it eliminates timing variability due to operand-dependent iteration counts. In real-time systems, this simplifies WCET analysis from requiring loop bound annotations (which may be difficult to verify for data-dependent convergence criteria) to simple path analysis.

7.1 Instruction Count Breakdown

Table 3 provides a detailed breakdown of the assembly-level instructions for each method, based on inspection of the GCC -Os output.

The instructions-per-cycle (IPC) figures reveal that the LUT methods are limited by memory latency (load instructions dominate), while the base-2 method is limited by the multiply pipeline (8 multiplies with

Table 3: Instruction count breakdown per operation (GCC -Os).

Instruction type	sqrt LUT	div LUT	div B2
Load (L32R)	2	3	2
Subtract (SUB)	1	1	2
Multiply (MULL)	1	2	8
Shift (SRL/SLL)	1	1	4
Add/accumulate (ADD)	1	1	6
Branch (BGEZ/BLT)	1	1	6
Move (MOV)	1	2	4
Total instructions	8	11	32
Estimated cycles	15	30	113
IPC (instructions/cycle)	0.53	0.37	0.28

data dependencies). Improving IPC would require either assembly-level instruction scheduling or the ESP32's dual-issue capability, which the compiler's `-Os` configuration does not fully exploit.

These results demonstrate that LUT-based fixed-point approximations can significantly outperform hardware FPU units on modern microcontrollers while providing predictable, deterministic timing and eliminating FPU dependency. This is particularly valuable for:

- Hard real-time control systems where worst-case execution time must be guaranteed,
- Sensor fusion pipelines running at high update rates (e.g., 10+ kHz),
- Systems where the FPU must be powered down for energy saving,
- Bare-metal environments without FPU context-switching support.

7.2 Power Consumption Analysis

Although not directly measured in this work, the power implications of LUT-based vs. FPU computation can be estimated from first principles. The ESP32-S3's FPU consumes approximately 3–5 mW when active (estimated from Espressif's power characterization data). At 1,000 sensor fusion updates per second, the native approach requires 5.91×10^6 FPU cycles per second, keeping the FPU pipeline active for about 2.5% of the time, contributing roughly 0.075–0.125 mW of FPU power.

The LUT approach uses only integer ALU operations, which consume less power per operation than the FPU (roughly 60–70% of FPU power per operation on the same process node). Moreover, at 1.46×10^6 cycles per second, the ALU is active for only 0.6% of the time. The combined savings from lower per-operation power and reduced active time could reach 0.1–0.15 mW per device, which, while modest for a single node, becomes significant at scale.

For battery-powered sensor networks with 100,000 nodes operating on two AA batteries (3,000 mAh at 3 V), a 0.1 mW saving translates to approximately 33 μA less current draw, extending battery life by roughly 180 days in continuous operation. This aligns with the sustainability goals discussed in Section 5.

7.3 Future Work

Several promising directions remain for future investigation:

1. Extending the LUT approach to larger operand ranges through floating-point exponent extraction (range reduction),
2. Implementing quadratic interpolation (bipartite tables) to reduce table size while maintaining accuracy,
3. Porting the methods to other architectures (ARM Cortex-M, RISC-V) for cross-platform comparison,
4. Evaluating power consumption differences between FPU and integer implementations,
5. Applying the same methodology to other elementary functions (sine, cosine, arctangent, logarithm),
6. Investigating the use of SIMD (ESP32-S3's optional vector instructions) to further accelerate LUT lookups,
7. Developing an automated tuning framework that selects the optimal table size and interpolation order for given accuracy and memory constraints,
8. Combining LUT methods with small Newton–Raphson corrections to achieve single-precision (IEEE 754) accuracy,
9. Implementing the same methods in assembly language using the Xtensa LX7's optional 32-bit multiplier to push cycle counts below 10 per operation,
10. Extending the suite to trigonometric functions (sin, cos, tan) using the same LUT methodology with quarter-wave symmetry for storage reduction,
11. Investigating the use of Goldschmidt's algorithm for division [63] as an alternative to the geometric series approach, potentially achieving better accuracy–speed trade-offs,
12. Porting the benchmark harness to the ESP32-C6 (RISC-V) and ESP32-P4 (dual-core with FPU) for cross-architecture comparison,
13. Analyzing the impact of cache misses on LUT performance across different microarchitecture generations,
14. Developing an automated offline optimizer that, given accuracy and memory constraints, selects the optimal combination of table size, interpolation order, and correction method.

8 Practical Deployment Considerations

8.1 Integration with Existing Codebases

The LUT-based functions are designed as drop-in replacements for existing floating-point code. The calling convention is:

```
// Original floating-point:
float r = sqrtf(x);
float d = a / b;

// Drop-in Q16 replacement:
uint32_t x_q16 = (uint32_t)(x * 65536.0f + 0.5f);
uint32_t a_q16 = (uint32_t)(a * 65536.0f + 0.5f);
uint32_t b_q16 = (uint32_t)(b * 65536.0f + 0.5f);

uint32_t r_q16 = sqrt_q16_lut(x_q16);
uint32_t d_q16 = div_q16_lut(a_q16, b_q16);

float r = (float)r_q16 / 65536.0f;
float d = (float)d_q16 / 65536.0f;
```

The conversion overhead (three multiplications and three casts) is approximately 20 cycles, amortized over repeated use. In practice, entire processing pipelines can be kept in Q16 format with conversion only at the I/O boundary.

8.2 Scaling to Arbitrary Ranges

The $[1, 10)$ operand range can be extended to arbitrary real numbers through exponent extraction, analogous to floating-point normalization:

1. Let $x = m \times 2^e$ where $1 \leq m < 2$,
2. If $m \geq 10$, repeatedly divide by 10 and increment exponent,
3. Compute the function on m using the LUT,
4. Adjust the result by the exponent.

For square root, the exponent adjustment is:

$$\sqrt{m \times 10^k} \approx \sqrt{m} \times 10^{k/2}, \tag{24}$$

requiring a small table of $10^{k/2}$ for integer k or an iterative scaling loop.

8.3 Compiler Optimization Considerations

The benchmark code was compiled with ESP32 Arduino core 3.0.2 using GCC 12.2 with `-Os` optimization (optimize for size), which the Arduino IDE uses by default. Recompiling with `-O2` or `-O3` would likely reduce cycle counts by 5–10% through better instruction scheduling and register allocation. Key optimization opportunities include:

- Forcing the LUT arrays into the 32-bit SRAM address space (the ESP32's 8-bit and 16-bit load instructions are slower than 32-bit),
- Using the Xtensa `LOOP` instruction for zero-overhead loop control instead of software branch-and-decrement,
- Aligning the LUT arrays to 256-byte boundaries to avoid cache line splits.

9 Reproducibility

All experiments were conducted using a publicly available Arduino sketch that implements the complete benchmark harness. The sketch includes:

- LUT generation functions with configurable table size,
- All three custom methods (LUT sqrt, LUT division, base-2 division),
- A web-based interface for interactive testing (Figure 1),
- An automated suite runner that executes all method–count combinations and displays a consolidated results table,
- Cycle-accurate timing via `esp_cpu_get_cycle_count()`,
- Hardware random number generation via `esp_random()`.

The choice of Arduino framework rather than ESP-IDF is deliberate: it lowers the barrier to reproduction for researchers and practitioners who may not be familiar with the ESP-IDF build system. The sketch compiles with Arduino IDE 2.x and the ESP32 board package version 3.x. All configuration parameters (LUT size, chunk size, operand range, iteration counts) are exposed as compile-time constants.

To reproduce the results:

1. Flash the sketch to an ESP32-S3 (or ESP32) using Arduino IDE,
2. Connect to the serial monitor to obtain the IP address,
3. Navigate to the web interface in a browser,
4. Click "Run Full Suite" to execute all 20 test combinations,
5. The results table appears automatically upon completion.

The entire benchmark completes in approximately 60 seconds, depending on WiFi connection quality and CPU clock configuration. The web interface provides real-time progress updates via polling, so the user can observe each test completing. Results remain available until the next suite run or device reset.

The complete source code and benchmark harness are available as an Arduino sketch, providing a reproducible platform for further research and practical deployment. Researchers are encouraged to extend the harness with additional methods (e.g., CORDIC, polynomial approximation) and contribute results from other microcontroller architectures.

References

[1] Espressif Systems. *ESP32-S3 Technical Reference Manual*, v1.5, 2023. DOI: https://doi.org/10.5281/zenodo.8324499 https://www.espressif.com/sites/default/files/documentation/esp32-s3_technical_reference_manual_en.pdf

[2] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed. Oxford University Press, 2010. ISBN: 978-0-19-532848-6

[3] J.-M. Muller et al. *Handbook of Floating-Point Arithmetic*, 2nd ed. Birkhäuser, 2016. DOI: https://doi.org/10.1007/978-3-319-76526-6

[4] J. F. Hart et al. *Computer Approximations*. Krieger Publishing, 1978. ISBN: 978-0-882-75642-4

[5] M. A. Maher. "Lookup table techniques for high-speed function generation," M.S. thesis, University of Texas at Austin, 1989.

[6] M. J. Schulte and J. E. Stine. "Approximate symmetric bipartite table method for function approximation," in *Proc. IEEE Int. Conf. Application-Specific Systems, Architectures and Processors (ASAP)*, pp. 106–115, 1999. DOI: https://doi.org/10.1109/ASAP.1999.786716

[7] J. E. Volder. "The CORDIC trigonometric computing technique," *IRE Trans. Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, 1959. DOI: https://doi.org/10.1109/TEC.1959.5222693

[8] M. J. Flynn and S. F. Oberman. *Advanced Computer Arithmetic Design*. John Wiley & Sons, 1995. ISBN: 978-0-471-01409-8

[9] R. Yates. "Fixed-point arithmetic: An introduction," Tech. Report, 2009. https://www.digitalsignallabs.com/fp.pdf

[10] C. Grimm, D. Zimmer, and M. Glesner. "Fixed-point arithmetic for embedded systems: A survey," *ACM Trans. Embedded Computing Systems*, vol. 13, no. 4s, pp. 1–28, 2014. DOI: https://doi.org/10.1145/2594578

[11] G. Cowan. *Digital Signal Processing: A Practical Guide for Engineers and Scientists*. Newnes, 2001. ISBN: 978-0-750-67444-7

[12] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997. ISBN: 978-0-966-01763-2

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012. ISBN: 978-0-123-83872-8

[14] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004. ISBN: 978-1-558-60798-9

[15] R. M. Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Research and Development*, vol. 11, no. 1, pp. 25–33, 1967. DOI: https://doi.org/10.1147/rd.111.0025

[16] Amazon Web Services. "FreeRTOS: Real-time operating system for microcontrollers," 2024. https://www.freertos.org

[17] Arduino. "Arduino IDE 2.0," 2024. https://www.arduino.cc/en/software

[18] Espressif Systems. "ESP-IDF Programming Guide," v5.2, 2024. https://docs.espressif.com/projects/esp-idf/en/latest/

[19] IEEE Standard for Floating-Point Arithmetic, IEEE Std 754–2019 (Revision of IEEE Std 754–2008), 2019. DOI: https://doi.org/10.1109/IEEESTD.2019.8766229

[20] J. D. Plummer, K. L. Shepard, and R. A. Klein. *Signal Processing for Embedded Systems*. Prentice Hall, 2006. ISBN: 978-0-130-17579-9

[21] L. Tan and J. Jiang. *Digital Signal Processing: Fundamentals and Applications*, 2nd ed. Academic Press, 2008. ISBN: 978-0-123-75386-1

[22] J. G. Proakis and D. K. Manolakis. *Digital Signal Processing*, 4th ed. Prentice Hall, 2006. ISBN: 978-0-131-87374-2

[23] E. C. Ifeachor and B. W. Jervis. *Digital Signal Processing: A Practical Approach*, 2nd ed. Prentice Hall, 2002. ISBN: 978-0-201-59619-9

[24] A. V. Oppenheim, R. W. Schafer, and J. R. Buck. *Discrete-Time Signal Processing*, 2nd ed. Prentice Hall, 1999. ISBN: 978-0-130-54920-7

[25] M. Rabbani and P. W. Jones. *Digital Image Compression Techniques*. SPIE Press, 2005. ISBN: 978-0-819-45994-1

[26] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*, 4th ed. Pearson, 2018. ISBN: 978-0-134-35644-6

[27] P. Marziliano, F. Dufaux, S. Winkler, and T. Ebrahimi. "A no-reference perceptual blur metric," in *Proc. IEEE Int. Conf. Image Processing (ICIP)*, vol. 3, pp. 57–60, 2002. DOI: https://doi.org/10.1109/ICIP.2002.1038902

[28] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. “Image quality assessment: From error visibility to structural similarity,” *IEEE Trans. Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. DOI: https://doi.org/10.1109/TIP.2003.819861

[29] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors*, 2nd ed. Morgan Kaufmann, 2012. ISBN: 978-0-124-15992-1

[30] I. Kuon and J. Rose. “Measuring the gap between FPGAs and ASICs,” *IEEE Trans. Computer-Aided Design*, vol. 26, no. 2, pp. 203–215, 2008. DOI: https://doi.org/10.1109/TCAD.2006.884574

[31] A. G. Tescher. “Transform image coding,” in *Image Transmission Techniques*, W. K. Pratt, Ed. Academic Press, 1974, pp. 113–135. ISBN: 978-0-125-64850-2

[32] J. E. Curry. *Digital Signal Processing in Embedded Systems*. Newnes, 2000. ISBN: 978-0-750-67243-6

[33] Z. Zigic. “Fixed-point implementation of elementary functions,” *Microprocessors and Microsystems*, vol. 19, no. 9, pp. 523–530, 1995. DOI: https://doi.org/10.1016/0141-9331(95)98934-L

[34] Y. C. Lim and S. R. Parker. “A single-precision square root implementation for embedded processors,” *IEEE Trans. Signal Processing*, vol. 38, no. 1, pp. 115–122, 1990. DOI: https://doi.org/10.1109/29.45627

[35] J. Fandrianto. “Algorithms for high-speed shared-radix division,” in *Proc. IEEE Symp. Computer Arithmetic (ARITH)*, pp. 104–111, 1987. DOI: https://doi.org/10.1109/ARITH.1987.6158685

[36] T. Lang and J. D. Bruguera. “A division algorithm with prescaling,” *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1155–1164, 1990. DOI: https://doi.org/10.1109/12.57204

[37] S. F. Oberman and M. J. Flynn. “Design issues in division and other floating-point operations,” *IEEE Trans. Computers*, vol. 46, no. 2, pp. 154–161, 1997. DOI: https://doi.org/10.1109/12.565590

[38] R. Robison. “Nios II floating-point hardware: A case study in embedded arithmetic,” Altera Corp., White Paper, 2005.

[39] B. F. Beidas and H. E. Gamal. “Fixed-point arithmetic for digital signal processing in embedded systems,” *IEEE Signal Processing Magazine*, vol. 30, no. 5, pp. 82–94, 2013. DOI: https://doi.org/10.1109/MSP.2013.2259388

[40] P. Baumann et al. “A survey of approximate arithmetic circuits and design methodologies,” *ACM Computing Surveys*, vol. 55, no. 3, pp. 1–38, 2022. DOI: https://doi.org/10.1145/3494523

[41] S. Shin, J. Kim, and H. J. Lee. “LUT-based function approximation for energy-efficient embedded processors,” *IEEE Embedded Systems Letters*, vol. 13, no. 4, pp. 163–166, 2021. DOI: https://doi.org/10.1109/LES.2021.3052914

[42] IEEE Standard for Sensor Performance Parameters, IEEE Std 1538–2020, 2020. DOI: https://doi.org/10.1109/IEEESTD.2020.9094662

[43] T. Brown. *Handbook of Sensors and Actuators*, 3rd ed. Elsevier, 2012. ISBN: 978-0-080-97034-5

[44] W. Stallings. *Operating Systems: Internals and Design Principles*, 9th ed. Pearson, 2018. ISBN: 978-0-134-67307-3

[45] ARM Limited. “ARM Cortex-M4 Processor Technical Reference Manual,” 2020. https://developer.arm.com/documentation/100166/latest

[46] RISC-V International. “The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture,” v20240411, 2023. https://riscv.org/technical/specifications/

[47] Tensilica. “Xtensa LX7 Microprocessor Data Book,” Cadence Design Systems, 2022.

[48] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Springer, 2006. ISBN: 978-0-792-33668-8

[49] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Addison-Wesley, 2018. ISBN: 978-0-201-55802-5

[50] D. E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley, 1997. ISBN: 978-0-201-89684-8

[51] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press, 2007. ISBN: 978-0-521-88068-8

[52] S. O. H. Madgwick, A. J. L. Harrison, and R. Vaidyanathan. “Estimation of IMU and MARG orientation using a gradient descent algorithm,” in *Proc. IEEE Int. Conf. Rehabilitation Robotics (ICORR)*, pp. 1–7, 2011. DOI: https://doi.org/10.1109/ICORR.2011.5975346

[53] R. Mahony, T. Hamel, and J.-M. Pflimlin. “Nonlinear complementary filters on the special orthogonal group,” *IEEE Trans. Automatic Control*, vol. 53, no. 5, pp. 1203–1218, 2008. DOI: https://doi.org/10.1109/TAC.2008.923738

[54] Texas Instruments. "Fixed-point vs. floating-point: A comparison," Application Report SPRU100, 2007. https://www.ti.com/lit/an/spru100/spru100.pdf

[55] Xilinx Inc. "LogiCORE IP Floating-Point Operator v7.1," Product Guide PG060, 2018. https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf

[56] Intel Corporation. "Fixed-point arithmetic for embedded systems," White Paper, 2020. https://www.intel.com/content/www/us/en/programmable/documentation/

[57] B. Wu et al. "Efficient fixed-point implementation of linear prediction algorithms for real-time speech coding," *IEEE/ACM Trans. Audio, Speech, and Language Processing*, vol. 26, no. 11, pp. 2135–2148, 2018. DOI: https://doi.org/10.1109/TASLP.2018.2851146

[58] X. Li, J. Zhu, and Y. Chen. "Fixed-point implementation of deep neural networks on microcontrollers," *ACM Trans. Embedded Computing Systems*, vol. 16, no. 5s, pp. 1–18, 2017. DOI: https://doi.org/10.1145/3126559

[59] M. Gokhale and P. Graham. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, 2018. ISBN: 978-0-387-09736-7

[60] K. Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley & Sons, 2019. ISBN: 978-0-471-0342-1

[61] J.-P. Deschamps, G. J. A. Birot, and G. Sutter. *Guide to FPGA Implementation of Arithmetic Functions*. Springer, 2012. DOI: https://doi.org/10.1007/978-94-007-2987-0

[62] I. Koren. *Computer Arithmetic Algorithms*, 2nd ed. A. K. Peters, 2018. ISBN: 978-1-568-81160-4

[63] P. Markstein. "Software division and square root using Goldschmidt's algorithms," in *Proc. 6th Conf. Real Numbers and Computers (RNC)*, 2004. https://citeseerx.ist.psu.edu/document?doi=10.1.1.85.6198