Convergence Computing Method for Complex Math Functions on ARMv7

Eduardo Bassani and Aman Monga

Department of Computer Science, University of Victoria

SENG 440: Embedded Systems

Professor Mihai Sima

August 8, 2025

Abstract

This report details the design, implementation, and optimization of a fixed-point mathematical function library for ARMv7-A embedded processors, specifically targeting complex operations like logarithms, exponentials, and roots without an FPU. By employing the Convergence Computing Method (CCM), which relies on efficient bit-shifting and additions, we developed high-performance alternatives to traditional software emulation. Our methodology involved both manual C-code optimizations and leveraging compiler flags, with the -O1 optimization level proving most effective by utilizing ARMv7-A specific hardware features. This approach yielded significant speedups for functions like exp2 and cbrt (up to 3.35x and 3.10x respectively), while maintaining excellent precision. The project concludes by proposing a custom ADDRS instruction to further accelerate these critical computations, showcasing the transformative potential of hardware-software co-design in embedded systems.

Table of Contents

1. Introduction

A key challenge in the field of embedded systems is performing complex mathematical operations, like logarithms and square roots, on hardware that is intentionally simple and low-cost. While a desktop computer can solve these problems instantly with a powerful Floating-Point Unit (FPU), most embedded processors lack this luxury to save cost and power.

1.1. Motivation

Standard C library functions, such as those in math.h, are typically designed for processors with FPUs. When run on a simpler, FPU-less embedded processor, these functions must fall back on slow software emulation, often breaking down a single calculation into a long series of multiplications and additions. This can create severe performance bottlenecks, which are unacceptable in systems that need to react in real-time. This project is motivated by the need for high-performance mathematical functions that are tailored specifically for the constraints of embedded systems. Our goal is to implement an alternative approach that is fast on simple hardware.

1.2. Project Goal and Scope

The primary goal of this project is to design, implement, and optimize a library of fixed-point mathematical functions using the Convergence Computing Method (CCM). CCM is a class of algorithms perfectly suited for this task because it calculates complex functions using only the most basic and fastest CPU operations: addition, subtraction, and bit-shifting.

This project implements four functions: base-2 logarithm (log2), base-2 exponential (exp2), square root (sqrt), and cubic root (cbrt). The project follows a structured methodology:

- **Implementation**: Develop initial non-optimized versions of the four functions in C, using fixed-point arithmetic and the core CCM algorithm.

- **Optimization**: Analyze the assembly code generated by the compiler to identify inefficiencies. Then, manually optimize the C code using techniques such as register allocation hints, pointer-based memory access, and loop unrolling to improve performance.

- **Benchmarking and Analysis**: Test both the non-optimized and optimized functions for performance and accuracy. The results are compared against the standard C library and

analyzed across different compiler optimization levels (-O0, -O1, -O2, -O3) to find the most effective combination of manual and compiler-driven optimization.

1.3. Project Environment and Toolchain

All development, compilation, and testing for this project were conducted in a controlled and consistent environment to ensure reliable results.

- **System Architecture**: The target platform is a 32-bit ARMv7-A processor. This architecture is representative of a wide range of processors used in modern embedded systems.

- **Operating System and Virtualization**: Development was performed within a Fedora 29 Linux environment (kernel version 4.18.16). As the development machines were macOS-based, this environment was hosted in a virtual machine managed by UTM, which uses the QEMU emulator as its backend. This setup provided a stable and accurate ARM environment for the project.

- **Compiler**: The GNU Compiler Collection (GCC) version 8.2.1 was used as the C compiler. GCC was chosen because it is a widely-used, open-source standard in the embedded systems industry and came pre-installed in the Fedora environment.

This report will now proceed to detail the background of the CCM algorithm, the design of our functions, the specific C-code and compiler optimizations applied, and a thorough analysis of the final performance and accuracy results.

---

2. Background

2.1. The Challenge of Complex Math on Simple Hardware

Many critical fields, including signal processing and wireless communications, rely on mathematical functions like logarithms, exponentials, and roots. On a typical desktop computer, these calculations are handled effortlessly by a dedicated Floating-Point Unit (FPU), a specialized hardware component designed for high-speed and high-precision mathematics.

However, the world of embedded systems operates under a different set of rules. These systems are designed for specific tasks and must be cost-effective and power-efficient. To meet

these constraints, they often use simpler processors that lack an FPU. When these systems need to perform complex math, they must rely on software routines.

A common software approach, like a Taylor series expansion, breaks a function down into a long sequence of multiplications and additions. While this works, it's extremely slow on a processor that doesn't have hardware for multiplication, creating a major performance bottleneck. This leads to an important question: How can we perform essential, complex mathematical calculations quickly and efficiently on resource-constrained hardware?

2.2. The Convergence Computing Method (CCM)

The Convergence Computing Method (CCM) is an approach designed specifically to solve this problem. It is a class of iterative algorithms that calculates these complex functions using only the most basic operations available on any CPU: addition and bit-shifting. Because it avoids slow multiplication and division operations, it is an inexpensive and effective method for embedded systems.

This general "shift-and-add" technique is versatile and can be used to calculate logarithms, exponentials, square roots, cubic roots, and even higher-order roots. It provides a powerful way to achieve high performance for demanding mathematical tasks without needing expensive, power-hungry hardware.

---

3. Algorithm Design

3.1. The Core CCM Framework

While the mathematical goals of the log2, exp2, sqrt, and cbrt functions differ, they were all designed using a unified framework based on the Convergence Computing Method (CCM). This approach is built around a core iterative loop that relies exclusively on simple, hardware-friendly operations.

Every function implements a "test-and-update" loop that runs for a fixed number of iterations, corresponding to the number of fractional bits in the data format. This ensures a predictable execution time. In each iteration of the loop, the same logical sequence occurs:

- A tentative value (mu) is calculated for a primary working variable.
- A corresponding tentative result (phi, mu_sqrt, mu_cbrt) is calculated.

- A simple conditional check is performed on the tentative value mu (e.g., if (mu <= UNITY)).

If the condition passes, the tentative values are accepted and overwrite the main working variables. If the condition fails, the tentative values are discarded, and the algorithm proceeds to the next iteration. This structure is common across all four implementations. The key difference between them lies in how the tentative values are calculated and what condition is used for the check.

The efficiency of CCM comes from how it calculates the tentative values. Instead of performing slow multiplications, it uses a "shift-and-add" technique. All the algorithms are based on applying successive multiplicative factors of the form (1 + 2^(-i)). A multiplication by this factor can be implemented efficiently as M + (M >> i), a bit-shift followed by an addition.

To improve precision, a rounding technique was used. The shift-and-add operation was implemented as (M + (1U << (i - 1))) >> i. This adds a "rounding bit" equivalent to half of the divisor before the right-shift, which more accurately rounds the result to the nearest integer rather than just truncating it.

3.2. Fixed-Point Data Representation

When designing systems using fixed-point data representation, balancing the trade-off between range and precision is crucial for all inputs, outputs, and intermediate calculations. The following data formats all use a 32-bit word length, which is the native size for the target ARM processor.

Both log2 and exp2 functions utilize the Q2.30 (q2_30_t) format. This is a signed format with 2 integer bits and 30 fractional bits, allowing it to cover a range of approximately [−2.0,+2.0). This single format was chosen because it can safely represent all values throughout both functions. For log2, the input is in [0.5,1.0), with intermediate results staying within the Q2.30 format's range, and the final output within [−1.0,0). A signed format is essential here to represent the negative output of the logarithm. Conversely, for exp2, the input is in [0.0,1.0), with intermediate results staying within the Q2.30 format's range, and the final output within [1.0,2.0).

The sqrt function employs the UQ4.28 (uq4_28_t) format. This is an unsigned format with 4 integer bits and 28 fractional bits, providing a range of [0, 16). The algorithm is designed for an input range of [1.0,4.0), with intermediate results staying within the UQ4.28 format's range, and the final output within [1.0,2.0). Since all values are positive, an unsigned format is sufficient.

For the cbrt function, the UQ5.27 (uq5_27_t) format is used. This is an unsigned format with 5 integer bits and 27 fractional bits, allowing it to cover a range of [0, 32). It's designed for inputs

in the range of [1.0,8.0), with intermediate results staying within the UQ5.27 format's range, and the final output within [1.0,2.0).

---

4. C Code Optimizations

This section evaluates the assembly code generated by the log2_non_optimized function, identifies its primary inefficiencies, and details the subsequent modifications made to the C code to produce more optimal assembly. While the log2 function was used as the primary example, these optimization techniques were applied across all four functions (log2, exp2, sqrt, cbrt) where applicable, as they shared a similar structure. The goal of these changes was to reduce memory access, simplify calculations, and minimize loop overhead, resulting in significantly faster execution. For a detailed comparison of the non-optimized and optimized logarithm C code, refer to Appendix A.1, and for their corresponding assembly outputs, see Appendix B.1.

4.1. Analysis of Non-Optimized Assembly

A close look at the assembly code generated for the log2_non_optimized function revealed several areas for improvement. Without optimization hints, the compiler defaulted to a very literal translation of the C code, which led to performance bottlenecks.

The most significant issue was excessive memory access. The assembly code constantly saved (stored) and retrieved (loaded) variables from the stack, which is a region of RAM. For example, instructions like str r3, [fp, #-8] (store value in stack memory) and ldr r3, [fp, #-16] (load value from stack memory) were used repeatedly for variables like M, f, mu, phi, and the loop counter i. Accessing RAM is orders of magnitude slower than accessing the CPU's internal registers. An ideal implementation would keep these frequently used variables in registers for the entire duration of the function.

Another inefficiency was the complex array address calculation. The C code log2_tab[i] accessed an element from the log2_tab table. In assembly, this became a multi-step operation: the base address of the table was loaded, the index i was loaded, i was multiplied by 4 (since each table entry is a 4-byte integer), and the result was added to the base address to find the final memory location. This calculation happened in every single iteration of the loop, which was redundant.

Finally, the code contained repetitive in-loop calculations and high loop overhead. The expression (1U << (i - 1)) was re-calculated in every loop, and the loop itself required instructions to increment, compare, and branch the counter variable i on every pass. For a loop that ran nearly 30 times, this overhead added up.

4.2. Optimization 1: Using the register Keyword

To address the excessive memory access, the first and simplest modification was to provide a hint to the compiler about which variables were most frequently used.

- **Problem**: Variables were stored on the slow-to-access stack, requiring constant load and store operations.

- **Modification**: The register keyword was prefixed to the declarations of the most frequently accessed variables. This suggested to the compiler that they should be stored in fast CPU registers instead of RAM.

  ```
  /* Before */
  q2_30_t M = M_norm;
  q2_30_t f = 0;

  /* After */
  register q2_30_t M = M_norm;
  register q2_30_t f = 0;
  ```

- **Impact**: This change dramatically reduced stack access. The log2_optimized assembly showed that variables were now held in registers throughout the loop. This eliminated dozens of slow memory operations, leading to a major speed improvement.

4.3. Optimization 2: Pointer Increment for Table Access

To fix the inefficient array indexing, we changed how the log2_tab was accessed. This optimization was specific to the log2 and exp2 functions, as sqrt and cbrt did not use a lookup table.

- **Problem**: Calculating the memory address of log2_tab[i] from scratch in every loop iteration was slow.

- **Modification**: Instead of using an index i, a pointer tab was initialized to point to the beginning of log2_tab. Inside the loop, the pointer was simply dereferenced to get the value and then incremented to point to the next element (*tab++).

  ```
  /* Before */
  ```

```
phi = f - log2_tab[i];

/* After */
// Initialized before the loop
register const q2_30_t *tab = log2_tab;

// Inside the loop
phi = f - *tab++;
```

- **Impact**: This change replaced a complex address calculation (base + index * 4) with a simple pointer increment, which was more efficient.

4.4. Optimization 3: Replacing Shift Calculation with a Mask

The non-optimized functions recalculated a power of two, (1U << (i - 1)), in every iteration. This was replaced with a much simpler operation.

- **Problem**: Repeatedly calculating 2^(i-1) using a variable shift amount was inefficient.

- **Modification**: A mask variable was initialized to 0 before the loop. Inside the loop, this mask was used in place of the (1U << (i - 1)) expression. The mask was then updated for the next iteration by shifting it left by one bit (mask = (mask << 1) | 1), which is equivalent to multiplying by two.

  ```
  /* Before */
  // Inside the loop
  mu  = M + ((M + (1U << (i - 1))) >> i);

  /* After */
  // Initialized before the loop
  register uint32_t mask = 0;

  // Inside the loop
  mu  = M + ((M + mask) >> i);
  ...
  mask = (mask << 1) | 1;
  ```

- **Impact**: Updating the mask with mask <<= 1 was more efficient than the original code, which required the processor to first calculate the shift amount (i-1) and then perform the variable shift.

4.5. Optimization 4: Loop Unrolling

To minimize loop overhead, we implemented loop unrolling, evaluating factors of both two and three for each function. Ultimately, unrolling by a factor of three yielded the most significant performance improvements.

- **Problem**: The for loop control logic: incrementing a counter, comparing it to a limit, and branching consumed valuable CPU cycles on every single pass.

- **Modification**: The loop body was "unrolled" by a factor of three. This meant the work of three original iterations was now performed inside a single, new iteration. The loop counter was then incremented by three (i += 3).

  ```
  /* Before */
  for (int i = 1; i < Q2_30_FBITS; i++) {
      // One iteration's work
  }

  /* After */
  for (int i = 0; i < Q2_30_FBITS; i += 3) {
      // Iteration 1's work
      ...
      // Iteration 2's work
      ...
      // Iteration 3's work
      ...
  }
  ```

- **Impact**: This optimization reduced the number of times the loop's control logic was executed by a factor of three. For a loop running ~30 times, this saved a significant number of instructions.

---

5. Compilation and Assembly

5.1. Compilation Strategy

To properly evaluate the impact of the C-level code modifications, the initial analysis relied on compiling the code with the -O0 flag. This flag explicitly tells the compiler not to apply any optimizations. By generating a direct, literal translation of the C code into assembly, it was

possible to establish a clean baseline. This approach ensured that any performance improvements came directly from the manual changes made to the code, not from the compiler's own optimization algorithms. For the entire assembly with the -O0 flag, refer to Appendix B.1.

5.2. Assembly Analysis: From Non-Optimized to Optimized

Analyzing the assembly code for both versions of the log2 function shows exactly how the C-level changes translated into more efficient assembly code.

5.2.1. Variable Storage in Registers

The most significant improvement came from eliminating the constant traffic between the CPU and stack memory for frequently used variables.

- **Before (Non-Optimized)**: In the original version, key variables like M and f were constantly pushed to and reloaded from the stack within the loop, a very slow process.

```
str   r3, [fp, #-8] @ push M to stack
ldr   r3, [fp, #-8] @ reload M
str   r3, [fp, #-12] @ push f
ldr   r3, [fp, #-12] @ reload f
```

- **After (Optimized)**: The register keyword successfully hinted to the compiler to keep these "hot" variables in fast CPU registers for the duration of the loop, eliminating stack traffic entirely.

```
mov   r10, r8 @ keep M in r10
mov   r7,  r9 @ keep f in r7
```

5.2.2. Table Access via Pointer Increment

The method for accessing the log2_tab lookup table was made far more direct.

- **Before (Non-Optimized)**: Accessing log2_tab[i] required a complex, multi-step address calculation in every loop cycle, involving loading the base address and the index i, then multiplying the index before adding it to the base.

```
movw  r3, #:lower16:log2_tab
movt  r3, #:upper16:log2_tab
ldr   r2, [fp, #-16] @ r2 = i
```

```
ldr   r3, [r3, r2, lsl #2] @ load log2_tab[i]
```

- **After (Optimized)**: The index calculation was replaced with a simple pointer. The code now performs a direct load from the pointer's address and increments the pointer by 4 bytes to prepare for the next iteration.

```
mov   r3, r5 @ r3 = tab pointer
add   r5, r3, #4 @ tab += 4
ldr   r3, [r3] @ phi = f – *old_tab
```

5.2.3. Replacing Dynamic Shift with a Running Mask

Instead of recalculating a shift value from the loop index i each time, the optimized code used a running mask that was updated with simple, fast instructions.

- **Before (Non-Optimized)**: The assembly had to load i, subtract 1 from it, and then perform a variable shift, a sequence of multiple instructions.

```
ldr   r3, [fp, #-16] @ r3 = i
sub   r3, r3, #1
mov   r2, #1
lsl   r2, r2, r3 @ r2 = 1 << (i−1)
```

- **After (Optimized)**: Each iteration's mask was efficiently updated with just one shift and one OR instruction, a much faster alternative.

```
lsl   r3, r4, #1 @ r3 = mask << 1
orr   r4, r3, #1 @ mask = r3 | 1
```

5.2.4. Reduced Loop Overhead (Loop Unrolling)

Finally, loop unrolling cut down on the non-essential bookkeeping instructions.

- **Before (Non-Optimized)**: The loop control logic: comparing the index i and branching was executed on every single pass of the loop.

```
ldr   r3, [fp, #-16] @ r3 = i
cmp   r3, #29
ble   .L5 @ branch each iteration
```

- **After (Optimized)**: By performing the work of three iterations inside a single pass, the number of branches was cut by two-thirds, reducing the overhead of the loop itself.

```
add   r6, r6, #3 @ i += 3
```

```
cmp   r6, #29
ble   .L12 @ only one branch per three original iterations
```

5.3. Comparison with Compiler Optimizations

After verifying that the manual C-level changes produced better assembly at -O0, the next step was to test their effectiveness against the compiler itself. To do this, both the non-optimized and the manually optimized versions of the code were re-compiled with standard optimization flags: -O1, -O2, and -O3. This experiment was designed to determine if a modern compiler, with its advanced optimization strategies, could achieve the same results or do an even better job than the manual code tuning.

The -O1 flag enabled the compiler to use its knowledge of the target ARM architecture to make powerful transformations that were not possible with C-level changes alone. For the entire assembly with the -O1 flag, refer to Appendix B.2.

5.3.1. Eliminating Branches with Conditional Execution

The most impactful change was the elimination of conditional branches inside the loop. An if statement in C normally compiles to a compare instruction followed by a branch instruction. Branches can be slow because they can disrupt the CPU's instruction pipeline. The -O1 compiler replaced this pattern with predication, or conditional execution.

- -O0 Assembly (with branch):

  ```
  cmp     r8, #1073741824
  bgt     .L9
  ```

- -O1 Assembly (no branch):

  ```
  cmp     ip, #1073741824
  suble   r0, r0, lr @ Execute only if Less than or Equal
  movle   r2, ip @ Execute only if Less than or Equal
  ```

5.3.2. Using the Barrel Shifter

The ARM architecture features a barrel shifter, which can perform a shift operation and another arithmetic operation in a single CPU cycle. The -O1 compiler took full advantage of this.

- **-O0 Assembly (separate instructions)**: A calculation involving a shift and an add required at least two separate instructions.

```
lsr     r3, r3, r6 @ Shift instruction
...
add     r3, r3, r2 @ Add instruction
```

- **-O1 Assembly (single instruction)**: The compiler merged these into a single command. This line of assembly shifts the value in ip and adds it to r2 in one cycle.

```
add     ip, r2, ip, lsr r1
```

5.3.3. Efficient Table Addressing

The -O1 compiler automatically implemented a pointer-increment strategy, but it adapted it intelligently for the unrolled loop in log2_optimized. Instead of incrementing after every access, it used a base pointer with fixed offsets. Only after all three values are used is the base pointer incremented by 12 bytes to point to the next block.

- -O1 Assembly:

```
ldr     lr, [r4, #-12] @ Load first value from base pointer + offset
...
ldr     ip, [r4, #-8] @ Load second value
...
ldr     lr, [r4, #-4] @ Load third value
...
add     r4, r4, #12 @ Increment base pointer once for all three
```

In conclusion, while the manual C-level optimizations resulted in significant speedups over the -O0 baseline, the -O1 compiler optimizations proved even more effective. By leveraging specific hardware features like conditional execution and the barrel shifter, the compiler generated code that was faster and more efficient than what could be achieved by manually tuning the C code alone.

---

6. Results

The functions were evaluated for performance and accuracy using a dedicated testbench. This testbench timed our functions against the standard C math library over 100,000 runs per input (to minimize the overhead of the timing mechanism itself) and calculated the error against the library's "golden" results. The tests were repeated across compiler optimization levels -O0, -O1, -O2, and -O3.

6.1. Performance Analysis

The results in Table 1 show that both our manual code changes and the compiler's optimizations had a significant impact on performance.

Table 1
Speedup Relative to C Library Under Different Compiler Optimization Levels

Function	Variant	–O0 vs C-lib	–O1 vs C-lib	–O2 vs C-lib	–O3 vs C-lib
log2	Non-Optimized	0.40×	1.22×	1.07×	1.19×
	Optimized	1.07×	1.27×	0.98×	0.97×
exp2	Non-Optimized	0.84×	2.44×	1.99×	2.32×
	Optimized	2.29×	2.59×	2.30×	3.35×
sqrt	Non-Optimized	0.16×	0.64×	0.61×	0.61×
	Optimized	0.63×	0.71×	0.65×	0.64×
cbrt	Non-Optimized	0.73×	2.86×	2.80×	2.80×
	Optimized	2.82×	3.10×	2.56×	2.54×

6.1.1. Impact of Manual Optimizations at -O0

To see the raw benefit of our C-level changes, we can compare the Non-Optimized and Optimized versions when compiled with no compiler help (-O0). For the cbrt function, our manual changes alone provided a 3.86× speedup (from 0.73x to 2.82x relative to the C library). Likewise, the log2 function saw a 2.67× speedup. This proves that our manual efforts to reduce memory access, unroll loops, and use efficient pointers were highly effective on their own.

6.1.2. The "Sweet Spot" of Compiler Optimization: -O1

The data clearly shows that the -O1 optimization flag consistently delivered the best results. For example, our optimized cbrt function peaked at a 3.10× speedup over the C library at -O1. Interestingly, performance often decreased at higher levels like -O2 and -O3. This is likely because more aggressive optimizations can sometimes increase code size and cause negative effects like CPU cache misses. Therefore, -O1 provided the most powerful and reliable set of optimizations for this project.

Overall, our exp2 and cbrt functions were a major success, achieving peak speedups of 3.35× and 3.10× respectively. Our sqrt implementation, however, did not outperform the standard library, likely due to highly specialized sqrt code that leverages dedicated hardware instructions found in standard math libraries.

6.2. Accuracy Analysis

Table 2 shows the accuracy of our functions at the optimal -O1 level. The errors are extremely small, on the order of 10^{-8} to 10^{-9} (one part in 100 million to one part in a billion).

Table 2
Maximum and Mean Absolute Errors at –O1 Optimization Level

Function	Variant	Max Error	MAE
log2	Non-Optimized	2.566148e-09	1.137015e-09
	Optimized	8.313117e-09	3.545866e-09
exp2	Non-Optimized	3.875694e-09	1.526405e-09
	Optimized	4.556943e-09	2.131167e-09
sqrt	Non-Optimized	1.321671e-08	5.877173e-09
	Optimized	1.321671e-08	5.877173e-09
cbrt	Non-Optimized	2.645114e-08	1.145755e-08
	Optimized	1.703820e-08	8.068383e-09

This level of precision is excellent and more than satisfactory for most embedded systems applications, where its impact would be negligible. Crucially, a comparison between the Non-Optimized and Optimized variants shows that we can successfully increase execution speed without sacrificing mathematical correctness.

---

7. Custom Instruction for Hardware Acceleration

While C-level and compiler optimizations provide significant speedups, most performance gains in embedded systems often come from custom hardware. To explore this, a new, hypothetical 32-bit instruction is proposed for the ARMv7 architecture. This instruction is designed to execute

the most demanding part of the iterative loop, the rounded shift and addition, in a single clock cycle.

7.1. Analysis of the Computational Bottleneck

In our optimized C code, we replaced the (1U << (i-1)) calculation with a running mask variable. While this improved performance over the non-optimized code, its implementation still represents the main bottleneck.

An analysis of the assembly generated for log2_optimized with the -O1 flag shows that the calculation of mu and the subsequent update of the mask variable require a total of four separate instructions within each step of the unrolled loop:

```
; 2 instructions to calculate mu using the mask
add     ip, r2, r3          ; ip = M + mask
add     ip, r2, ip, lsr r1  ; mu = M + (ip >> i)

; 2 instructions to update the mask for the next step
lsl     r3, r3, #1          ; mask = mask << 1
orr     r3, r3, #1          ; mask = mask | 1
```

This four-instruction sequence is a computationally intensive part of the loop. A custom instruction that performs the entire mathematical step at once would provide a substantial speedup.

7.2. Instruction Definition: ADDRS

To accelerate the CCM algorithm, we define the Add with Rounded Shift (ADDRS) instruction. This instruction is designed to implement the original mathematical operation M + ((M + (1U << (i - 1))) >> i) directly in hardware, making the software-based mask variable completely unnecessary.

7.2.1. Syntax

ADDRS{cond} Rd, Rn, Rm

- **{cond}**: The standard ARM conditional execution code (e.g., EQ, NE).
- **Rd**: The destination register, which will hold the final result (mu).

- **Rn**: The first source register, which holds the input value (M).
- **Rm**: The second source register, which holds the shift amount (i).

7.2.2. Semantics

The ADDRS instruction performs the following operation in a single step:

Rd = Rn + ((Rn + (1 << (Rm - 1))) >> Rm)

7.3. Impact on Performance

Integrating the ADDRS instruction would fundamentally change the performance of the CCM functions by replacing the entire multi-instruction sequence for calculating mu and updating the mask.

- Before (Manually Optimized C with -O1):

```
; 4 instructions to calculate mu and update mask
add     ip, r2, r3
add     ip, r2, ip, lsr r1
lsl     r3, r3, #1
orr     r3, r3, #1
```

- After (with new ADDRS instruction):

```
; 1 instruction to calculate mu directly
; Assume r1=M, r2=i, and r3=mu
ADDRS r3, r1, r2
```

This change replaces a four-instruction block with a single instruction, resulting in a 4× speedup for that critical part of the algorithm. This improvement has a cascading effect, especially in the unrolled loops of the log2 and exp2 functions. Since their loops run for 10 iterations with the calculation performed three times per iteration, the ADDRS instruction would replace a total of 120 instructions (10 iterations × 3 blocks/iteration × 4 instructions/block) with just 30 instructions (10 × 3 × 1). This yields a net reduction of 90 instructions from the function's core logic.

The benefit of this custom instruction would be even more pronounced for the sqrt and cbrt functions. The sqrt algorithm uses three shift-and-add sequences per step, while cbrt uses four. A hardware-based ADDRS instruction would consolidate these repeated calculations even more

effectively, leading to a greater overall performance boost than what is seen in log2 and exp2. This highlights the transformative potential of hardware-software co-design, which can deliver efficiency gains far beyond what software-only optimizations can achieve.

---

8. Conclusion

This project set out to address the critical challenge of performing complex mathematics on resource-constrained embedded processors by designing, implementing, and optimizing a library of fixed-point functions using the Convergence Computing Method. The results conclusively demonstrate that CCM is a highly effective strategy for this task, capable of delivering significant performance improvements over standard library functions without the need for a hardware FPU.

Our investigation into performance yielded several key insights. First, manual C-code optimizations provided a substantial performance boost on their own, improving execution speed by up to 3.86× at the -O0 compilation level. However, the most powerful results were achieved by combining these manual efforts with compiler optimizations. The -O1 flag proved to be the "sweet spot," as it enabled the compiler to leverage hardware-specific features of the ARMv7 architecture, such as conditional execution and the barrel shifter, to generate exceptionally efficient assembly code that often outperformed more aggressive levels like -O2 and -O3.

Ultimately, our optimized exp2 and cbrt functions were a clear success, achieving peak speedups of 3.35× and 3.10×, respectively, over the standard C math library. While our sqrt implementation did not surpass the library's performance, likely due to highly specialized hardware instructions available to standard implementations, the overall methodology proved sound. Crucially, these performance gains did not compromise correctness. All functions maintained excellent precision, proving that the CCM approach is both fast and reliable.

Finally, our analysis of the remaining bottlenecks in the optimized assembly led to the design of a hypothetical instruction. This proposal highlights a clear path for future improvements at the micro-architectural level.

References

Arm Limited. (n.d.). *Arm architecture reference manual: ARMv7-A and ARMv7-R edition* (DDI 0406). Arm Developer. https://developer.arm.com/documentation/ddi0406/latest/ Arm Developer

Arm Limited. (n.d.). *Arm Cortex-A series (Armv7-A) programmer's guide* (DEN0013, 0400). Arm Developer. https://developer.arm.com/documentation/den0013/0400

Fixed-point arithmetic. (n.d.). Wikipedia. Retrieved August 8, 2025, from https://en.wikipedia.org/wiki/Fixed-point_arithmetic

Arar, S. (2017, November 30). *Fixed-Point representation: The Q format and addition examples*. All About Circuits. https://www.allaboutcircuits.com/technical-articles/fixed-point-representation-the-q-format-and-addition-examples/

Rothwell, T., & Youngman, J. (n.d.). The GNU C reference manual. GNU Project. Retrieved August 8, 2025, from https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

Hardware floating-point vs software floating-point. (2024, September 10). Embedded Insights. https://codewithivy.hashnode.dev/hardware-floating-point-vs-software-floating-point

OpenAI. (2025). *ChatGPT (o3)* [Large language model]. https://chat.openai.com/

Google. (2025). *Gemini (2.5 Pro)* [Large language model]. https://gemini.google.com/

Appendices

Appendix A: C Code

A.1 Logarithm Functions Code

```
#include "../include/q_formats.h"
#include "../include/log2_table.h"

q2_30_t log2_non_optimized(q2_30_t M_norm) {
    q2_30_t M = M_norm;
    q2_30_t f = 0;

    q2_30_t mu  = M + M;
    q2_30_t phi = f - log2_tab[0];

    if (mu <= Q2_30_ONE) {
        M = mu;
        f = phi;
    }

    for (int i = 1; i < Q2_30_FBITS; i++) {
        q2_30_t mu  = M + ((M + (1U << (i - 1))) >> i);
        q2_30_t phi = f - log2_tab[i];

        if (mu <= Q2_30_ONE) {
            M = mu;
            f = phi;
        }
    }

    return f;
}

q2_30_t log2_optimized(q2_30_t M_norm) {
    register q2_30_t M = M_norm;
    register q2_30_t f = 0;
    register const q2_30_t *tab = log2_tab;
    register uint32_t mask = 0;

    register q2_30_t mu;
    register q2_30_t phi;
```

```
    for (register int i = 0; i < Q2_30_FBITS; i += 3) {
        mu  = M + ((M + mask) >> i);
        phi = f - *tab++;
        if (mu <= Q2_30_ONE) {
            M = mu;
            f = phi;
        }
        mask = (mask << 1) | 1;

        mu  = M + ((M + mask) >> (i + 1));
        phi = f - *tab++;
        if (mu <= Q2_30_ONE) {
            M = mu;
            f = phi;
        }
        mask = (mask << 1) | 1;

        mu  = M + ((M + mask) >> (i + 2));
        phi = f - *tab++;
        if (mu <= Q2_30_ONE) {
            M = mu;
            f = phi;
        }
        mask = (mask << 1) | 1;
    }

    return f;
}
```

Appendix B: Assembly

B.1 Logarithm Functions Assembly (-O0 flag)

```
        .arch armv7-a
        .eabi_attribute 28, 1
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 2
```

```
        .eabi_attribute 30, 6
        .eabi_attribute 34, 1
        .eabi_attribute 18, 4
        .file   "log.c"
        .text
        .section        .rodata
        .align  2
        .type   log2_tab, %object
        .size   log2_tab, 120
log2_tab:
        .word   1073741824
        .word   628098702
        .word   345667660
        .word   182455581
        .word   93912511
        .word   47667823
        .word   24017256
        .word   12055174
        .word   6039314
        .word   3022600
        .word   1512037
        .word   756203
        .word   378148
        .word   189085
        .word   94546
        .word   47274
        .word   23637
        .word   11819
        .word   5909
        .word   2955
        .word   1477
        .word   739
        .word   369
        .word   185
        .word   92
        .word   46
        .word   23
        .word   12
        .word   6
        .word   3
        .text
        .align  2
        .global log2_non_optimized
        .arch armv7-a
```

```
        .syntax unified
        .arm
        .fpu vfpv3-d16
        .type   log2_non_optimized, %function
log2_non_optimized:
        @ args = 0, pretend = 0, frame = 40
        @ frame_needed = 1, uses_anonymous_args = 0
        @ link register save eliminated.
        str     fp, [sp, #-4]!
        add     fp, sp, #0
        sub     sp, sp, #44
        str     r0, [fp, #-40]
        ldr     r3, [fp, #-40]
        str     r3, [fp, #-8]
        mov     r3, #0
        str     r3, [fp, #-12]
        ldr     r3, [fp, #-8]
        lsl     r3, r3, #1
        str     r3, [fp, #-20]
        mov     r2, #1073741824
        ldr     r3, [fp, #-12]
        sub     r3, r3, r2
        str     r3, [fp, #-24]
        ldr     r3, [fp, #-20]
        cmp     r3, #1073741824
        bgt     .L2
        ldr     r3, [fp, #-20]
        str     r3, [fp, #-8]
        ldr     r3, [fp, #-24]
        str     r3, [fp, #-12]
.L2:
        mov     r3, #1
        str     r3, [fp, #-16]
        b       .L3
.L5:
        ldr     r3, [fp, #-16]
        sub     r3, r3, #1
        mov     r2, #1
        lsl     r2, r2, r3
        ldr     r3, [fp, #-8]
        add     r2, r2, r3
        ldr     r3, [fp, #-16]
        lsr     r2, r2, r3
        ldr     r3, [fp, #-8]
```

```
        add     r3, r2, r3
        str     r3, [fp, #-28]
        movw    r3, #:lower16:log2_tab
        movt    r3, #:upper16:log2_tab
        ldr     r2, [fp, #-16]
        ldr     r3, [r3, r2, lsl #2]
        ldr     r2, [fp, #-12]
        sub     r3, r2, r3
        str     r3, [fp, #-32]
        ldr     r3, [fp, #-28]
        cmp     r3, #1073741824
        bgt     .L4
        ldr     r3, [fp, #-28]
        str     r3, [fp, #-8]
        ldr     r3, [fp, #-32]
        str     r3, [fp, #-12]
.L4:
        ldr     r3, [fp, #-16]
        add     r3, r3, #1
        str     r3, [fp, #-16]
.L3:
        ldr     r3, [fp, #-16]
        cmp     r3, #29
        ble     .L5
        ldr     r3, [fp, #-12]
        mov     r0, r3
        add     sp, fp, #0
        @ sp needed
        ldr     fp, [sp], #4
        bx      lr
        .size   log2_non_optimized, .-log2_non_optimized
        .align  2
        .global log2_optimized
        .syntax unified
        .arm
        .fpu vfpv3-d16
        .type   log2_optimized, %function
log2_optimized:
        @ args = 0, pretend = 0, frame = 8
        @ frame_needed = 1, uses_anonymous_args = 0
        @ link register save eliminated.
        push    {r4, r5, r6, r7, r8, r9, r10, fp}
        add     fp, sp, #28
        sub     sp, sp, #8
```

```
        str     r0, [fp, #-32]
        ldr     r10, [fp, #-32]
        mov     r7, #0
        movw    r5, #:lower16:log2_tab
        movt    r5, #:upper16:log2_tab
        mov     r4, #0
        mov     r6, #0
        b       .L8
.L12:
        mov     r3, r10
        add     r3, r4, r3
        lsr     r3, r3, r6
        mov     r2, r10
        add     r3, r3, r2
        mov     r8, r3
        mov     r3, r5
        add     r5, r3, #4
        ldr     r3, [r3]
        sub     r9, r7, r3
        cmp     r8, #1073741824
        bgt     .L9
        mov     r10, r8
        mov     r7, r9
.L9:
        lsl     r3, r4, #1
        orr     r4, r3, #1
        mov     r3, r10
        add     r2, r4, r3
        add     r3, r6, #1
        lsr     r3, r2, r3
        mov     r2, r10
        add     r3, r3, r2
        mov     r8, r3
        mov     r3, r5
        add     r5, r3, #4
        ldr     r3, [r3]
        sub     r9, r7, r3
        cmp     r8, #1073741824
        bgt     .L10
        mov     r10, r8
        mov     r7, r9
.L10:
        lsl     r3, r4, #1
        orr     r4, r3, #1
```

```
        mov     r3, r10
        add     r2, r4, r3
        add     r3, r6, #2
        lsr     r3, r2, r3
        mov     r2, r10
        add     r3, r3, r2
        mov     r8, r3
        mov     r3, r5
        add     r5, r3, #4
        ldr     r3, [r3]
        sub     r9, r7, r3
        cmp     r8, #1073741824
        bgt     .L11
        mov     r10, r8
        mov     r7, r9
.L11:
        lsl     r3, r4, #1
        orr     r4, r3, #1
        add     r6, r6, #3
.L8:
        cmp     r6, #29
        ble     .L12
        mov     r3, r7
        mov     r0, r3
        sub     sp, fp, #28
        @ sp needed
        pop     {r4, r5, r6, r7, r8, r9, r10, fp}
        bx      lr
        .size   log2_optimized, .-log2_optimized
        .ident  "GCC: (GNU) 8.2.1 20180801 (Red Hat 8.2.1-2)"
        .section        .note.GNU-stack,"",%progbits
```

B.2 Logarithm Functions Assembly (-O1 flag)

```
        .arch armv7-a
        .eabi_attribute 28, 1
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 2
        .eabi_attribute 30, 1
        .eabi_attribute 34, 1
```

```
        .eabi_attribute 18, 4
        .file   "log.c"
        .text
        .align  2
        .global log2_non_optimized
        .arch armv7-a
        .syntax unified
        .arm
        .fpu vfpv3-d16
        .type   log2_non_optimized, %function
log2_non_optimized:
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 0, uses_anonymous_args = 0
        push    {r4, lr}
        mov     r1, r0
        lsl     r3, r0, #1
        cmp     r3, #1073741824
        movle   r1, r3
        cmp     r3, #1073741824
        movgt   r0, #0
        movle   r0, #-1073741824
        movw    ip, #:lower16:.LANCHOR0
        movt    ip, #:upper16:.LANCHOR0
        mov     r2, #1
        mov     r4, r2
.L4:
        sub     r3, r2, #1
        add     r3, r1, r4, lsl r3
        add     r3, r1, r3, lsr r2
        ldr     lr, [ip, #4]!
        cmp     r3, #1073741824
        suble   r0, r0, lr
        movle   r1, r3
        add     r2, r2, #1
        cmp     r2, #30
        bne     .L4
        pop     {r4, pc}
        .size   log2_non_optimized, .-log2_non_optimized
        .align  2
        .global log2_optimized
        .syntax unified
        .arm
        .fpu vfpv3-d16
        .type   log2_optimized, %function
```

```
log2_optimized:
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 0, uses_anonymous_args = 0
        push    {r4, r5, r6, lr}
        mov     r2, r0
        ldr     r4, .L17
        mov     r1, #0
        mov     r3, r1
        mov     r0, r1
.L13:
        add     ip, r2, r3
        add     ip, r2, ip, lsr r1
        mov     r5, r4
        ldr     lr, [r4, #-12]
        cmp     ip, #1073741824
        suble   r0, r0, lr
        movle   r2, ip
        lsl     r3, r3, #1
        orr     r3, r3, #1
        mov     r6, r1
        add     lr, r2, r3
        add     ip, r1, #1
        add     lr, r2, lr, lsr ip
        ldr     ip, [r4, #-8]
        cmp     lr, #1073741824
        suble   r0, r0, ip
        movle   r2, lr
        lsl     r3, r3, #1
        orr     r3, r3, #1
        add     ip, r2, r3
        add     lr, r1, #2
        add     ip, r2, ip, lsr lr
        ldr     lr, [r4, #-4]
        cmp     ip, #1073741824
        suble   r0, r0, lr
        movle   r2, ip
        lsl     r3, r3, #1
        orr     r3, r3, #1
        add     r1, r1, #3
        add     r4, r4, #12
        cmp     r1, #30
        bne     .L13
        pop     {r4, r5, r6, pc}
.L18:
```

```
        .align  2
.L17:
        .word   .LANCHOR0+12
        .size   log2_optimized, .-log2_optimized
        .section        .rodata
        .align  2
        .set    .LANCHOR0,. + 0
        .type   log2_tab, %object
        .size   log2_tab, 120
log2_tab:
        .word   1073741824
        .word   628098702
        .word   345667660
        .word   182455581
        .word   93912511
        .word   47667823
        .word   24017256
        .word   12055174
        .word   6039314
        .word   3022600
        .word   1512037
        .word   756203
        .word   378148
        .word   189085
        .word   94546
        .word   47274
        .word   23637
        .word   11819
        .word   5909
        .word   2955
        .word   1477
        .word   739
        .word   369
        .word   185
        .word   92
        .word   46
        .word   23
        .word   12
        .word   6
        .word   3
        .ident  "GCC: (GNU) 8.2.1 20180801 (Red Hat 8.2.1-2)"
        .section        .note.GNU-stack,"",%progbits
```