Automated design of floating-point logarithm functions on integer processors

Guillaume Revy

Automated design of floating-point logarithm functions on integer processors

Guillaume Revy
Univ. Perpignan Via Domitia, DALI, F-66860, Perpignan, France
Univ. Montpellier II, LIRMM, UMR 5506, F-34095, Montpellier, France
CNRS, LIRMM, UMR 5506, F-34095, Montpellier, France

***Abstract*—Nowadays the automated design of efficient floating-point implementations of correctly rounded elementary functions like $\cos$, $\sin$, $\log$, $\exp$, $\cdots$ is a real challenge. Indeed, the variety of hardware architectures and floating-point formats makes such implementation process tedious and error-prone. This article focuses on the particular case of floating-point $\log_b(x)$ functions on integer processors. First it proposes a unified range reduction for $\log_b(x)$, that enables to reduce the evaluation of these functions to a single well-chosen polynomial. Second it gives some sufficient conditions on the approximation and evaluation errors to guarantee correct rounding. And third it shows how to automate the implementation process on integer processors, when $b \in \{2, \exp(1), 10\}$. Finally we illustrate how this automated approach enables to speedup the design of efficient implementations of $\log_b(x)$ for standard floating-point formats.**

Keywords: **correctly rounded floating-point logarithm, polynomial evaluation, automated design, integer processor.**

I. Introduction

Since 2008, the IEEE 754 standard requires the correct rounding for the basic operations (namely, $+$, $-$, $\times$, $/$, and $\sqrt{}$) and for the four rounding direction attributes (rounding to-nearest-even, rounding upward, rounding downward, and rounding toward zero) [1]. This property is now also recommended for the elementary functions like $\cos$, $\sin$, $\log$, $\exp$, $\cdots$. In this sense, this article focuses on the automated design of correctly rounded implementations for the logarithm functions on integer processors, for the rounding to-nearest-even.

In the literature, many techniques have already been designed for the implementation of logarithm functions. They classically rely on table lookups and/or polynomial evaluations. In [2], Tang proposes a method to implement several functions including $\log(x)$, that combines table lookup and polynomial evaluation, and where tabulated values are regularly spaced values. It enables to achieve an output accuracy very close to 0.5 ulp. This technique has more recently been used for the design of logarithm function in the Intel libm [3], [4]. In Tang's method, the tabulated values are approximation of the function to be implemented. In order to increase the accuracy of these tabulated values, Gal proposes the *accurate tables method* to implement a set of elementary functions, especially $\log(x)$ and $\log_{10}(x)$ [5], [6]. The values are not regularly spaced anymore, but chosen so that they are closer to representable floating-point numbers. This improves the accuracy of the tabulated values of about 10 bits, and testings show that this enables to achieve correct rounding in about 99.9% of the test cases.

Both these methods are well-adapted for software and hardware implementations. However when targeting specific hardware, others techniques have been studied [7], [8], [9], [10]. In [7], [8], Wong and Goto propose a first method where tables addressed by $p/2$ bits are used for implementing functions for numbers of precision p. However this does not scale well when the precision p increases. These authors propose a second method in order to make the most of the underlying hardware circuit [11]. This relies on the use of 16×56-bit rectangular multipliers. Being faster than double precision multipliers, these enable to speedup the full implementation, while keeping an output accuracy of 1 ulp. More recently, this approach has been adapted in software to implement correctly rounded $\log(x)$ in the CR-Libm project [12].

All these methods rely on tabulated values. On the other side of this spectrum, another approach consists in using only polynomial evaluations. This is for example done in [13], which deals with hardware designs of correctly rounded implementations of the $\log_2(x)$ function for floating-point numbers of precision 16 and 24.

In this article, we extend to transcendental functions the work done in [14] on square root and division. Particularly our target architecture is the ST231 processor, a 4-issue 32-bit VLIW integer processor from STMicroelectronics, providing 1-cycle 32-bit ALU operations but a 3-cycle 32-bit multiplication. On this target, using table lookup methods may be particularly ill-suited, since accessing data from memory might lead to a penalty of about 100 cycles [15]. For these reasons, we choose to focus on methods based on polynomial evaluation, and relying on integer arithmetic. In addition, to avoid branches and to reduce the code size, unlike methods based on a two-step process (*quick* and *accurate* steps), our goal is to provide implementation relying on only one polynomial evaluation step enabling to reach the targeted accuracy.

Hence the main contributions of this article are:

- A unified range reduction for the $\log_b(x)$ functions, similarly to [3, § 11] but relying on one polynomial evaluation only instead of table lookups,
- Sufficient conditions on the polynomial approximation and evaluation accuracy to reach the required accuracy,
- Some guidelines on how to implement these logarithm functions on integer arithmetic, together with some elements on the error analysis of these implementations,
- And finally a script implementing this approach and enabling to automate the implementation of $\log_b(x)$ for various precisions p.

This article is organized as follows: After some preliminaries and notations in Section II, Section III details the algorithm we propose to implement $\log_b(x)$. Since this relies

Input x	NaN	$-\infty$	$x<0$	± 0	$x>0$	$+\infty$
Output	qNaN			$-\infty$	RN($\log_b(x)$)	$+\infty$

Table I: Special values of $\log_b(x)$.

on polynomial evaluation, Section IV gives some sufficient conditions on approximation and evaluation errors required in order to ensure a certain output accuracy, and thus to reach correct rounding. Then Section V is devoted to the implementation on integer processors, and its error analysis. Finally Section VI shows some experimental results, before concluding in Section VII.

II. Preliminaries and notations

Let us consider the following floating-point data: ± 0, $\pm\infty$, quiet or signaling Not-a-Numbers (qNaN, sNaN), or binary floating-point numbers x defined in [1] as:

$$x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x}, \tag{1}$$

with $s_x \in \{0,1\}$, $m_x = (m_0.m_1 \dots m_{p-1})_2$ with $m_i \in \{0,1\}$, and $e_x \in \{e_{\min}, \dots, e_{\max}\}$ encoding in at most w digits. The precision p and extremal exponents $e_{\min}$ and $e_{\max}$ are assumed to be integers such as $p \geq 2$ and $e_{\min} = 1 - e_{\max}$. Using these notations, let us define α and Ω, the smallest and largest non-zero positive floating-point number, respectively:

$$\alpha = 2^{e_{\min}-p+1} \qquad \text{and} \qquad \Omega = (2 - 2^{1-p}) \cdot 2^{e_{\max}}.$$

Now let x be a non-zero floating-point number. Its normalized representation is given as follows:

$$x = (-1)^s \cdot m \cdot 2^e, \tag{2}$$

with $m = m_x \cdot 2^\lambda$ and $e = e_x - \lambda$, and where $\lambda \in \{0, \cdots, p-1\}$ denotes the number of leading zeros in the binary representation of m_x. It follows that $e \in \{e_{\min} - p + 1, \cdots, e_{\max}\}$ and $m \in [1, 2 - 2^{1-p}]$.

Finally, let us denote by x^- and x^+ the predecessor and successor of the floating-point number x, respectively.

III. Algorithm for correctly rounded $\log_b(x)$

Here and hereafter, we assume the value b defined as follows:

$$b \in \mathbb{R}_+^* \quad \text{and} \quad b \in \{2, \exp(1), 10\}. \tag{3}$$

The rest of this section presents the unified range reduction we propose to implement $\log_b(x)$ for various precisions p.

A. Special and straightforward inputs of $\log_b(x)$

Let x be a floating-point number as in (1). From [1] we know that x is a *special input* for $\log_b(x)$ if $x \in \{\text{NaN}, -\infty, x < 0, \pm 0, +\infty\}$. Table I shows the result to be returned according to the input x. Otherwise if x is not a special input, the routine shall return the result RN($\log_b(x)$). This occurs when $x > 0$ and more particularly $\alpha \leq x \leq \Omega$.

Property 1. *Let x be a non-negative floating-point number as in* (1)*, with $\alpha \leq x \leq \Omega$. Since $\log_b(x)$ is a transcendental function, and both floating-point numbers and breakpoints are algebraic numbers, we know that $\log_b(x)$ cannot be a breakpoint, except for straightforward inputs such as $\log_b(1) = 0$. (See [16] for details.)*

It follows from Property 1 that, except for straightforward inputs, in rounding to-nearest-even, $\log_b(x)$ cannot be halfway between two floating-point numbers. The only straightforward input is $x = 1$: in this case, the $\log_b(x)$ routine shall return the floating-point value 0. In the rest of this section, we consider $\alpha \leq x \leq \Omega$ and $x \neq 1$.

Property 2. *Let x be a non-negative floating-point number as in* (1)*, with $\alpha \leq x \leq \Omega$ and $x \neq 1$. If $\alpha > b^{-\Omega}$ and $\Omega < b^\Omega$, then the function $x \mapsto \log_b(x)$ does not overflow.*

Proof: If $\alpha > b^{-\Omega}$ and $\Omega < b^\Omega$, we deduce that $b^{-\Omega} < x < b^\Omega$. Hence since $x \mapsto \log_b(x)$ is monotonically increasing over $\mathbb{R}_+^*$, it follows that $-\Omega < \log_b(x) < \Omega$, and more generally $|\log_b(x)| < \Omega$, which concludes the proof. ■

Property 3. *Let x be a non-negative floating-point number as in* (1)*, with $\alpha \leq x \leq \Omega$ and $x \neq 1$. If $2^{-p}/\log(b) > 2^{e_{min}}$, the function $x \mapsto \log_b(x)$ does not underflow.*

Proof: On a first hand, using Taylor expansion of the function $\log(1+x)$ around 0, we know that $\log(x^+) > 2^{-p}$, and since $\log(b) > 0$, $\log_b(x^+) > 2^{-p}/\log(b)$. On a second side, using Taylor expansion of the function $\log(1-x)$ around 0, we know that $\log(x^-) < -2^{-p}$, and $\log_b(x-) < -2^{-p}/\log(b)$. Since $\log_b(x)$ is monotonically increasing over $\mathbb{R}_+^*$, we have $|\log_b(x)| > 2^{-p}/\log(b)$. And if $2^{-p}/\log(b) > 2^{e_{\min}}$, then $|\log_b(x)| > 2^{e_{\min}}$, which concludes the proof. ■

Using Properties 2 and 3 we can conclude that, under certain conditions, the function $\log_b(x)$ does never overflow nor underflow. For the particular cases of IEEE 754 implementations, all these conditions are verified, and Properties 2 and 3 hold. This simplifies considerably the implementation process, as shown further in Section V.

B. Unified range reduction for $\log_b(x)$

Let us detail our unified range reduction: To do so, let x be a non-negative floating-point number as in (2) with $s = 0$, and such as $\alpha \leq x \leq \Omega$ and $x \neq 1$. It follows that:

$$\log_b(x) = \log_b(2) \cdot e + \log_b(m). \tag{4}$$

When the result gets close to 0, using this rewriting may lead to a catastrophic cancellation when both terms are of opposite sign. Since $\log_b(2)$ and $\log_b(m)$ are non-negative, as explained in [3], this may occur when $e = -1$. To remedy this, we rewrite (4) as

$$\log_b(x) = \begin{cases} \log_b(2) \cdot e + \log_b(m), & \text{if } m < 1.5, \\ \log_b(2) \cdot (e+1) + \log_b(m/2), & \text{otherwise,} \end{cases}$$

that is

$$\log_b(x) = \log_b(2) \cdot (e + \tau) + \log_b(m/2^\tau), \tag{5}$$

where $\tau = [m \geq 1.5]$. In addition, the case $m/2^\tau = 1$ can be handled separately, and is no longer considered in this section: indeed $\log_b(1) = 0$ and $\log_b(x) = \log_b(2) \cdot (e + \tau)$.

Let us now consider the general case $m/2^\tau \neq 1$. Notice that even in this case, $\log_b(m/2^\tau)$ may also get arbitrarily close to 0. And computing $\log_b(x)$ as in (5) may lead to a loss of

accuracy especially when $e+\tau=0$. Hence to overcome this issue, let us define $t = m/2^\tau - 1$ where

$$t \in \mathcal{T} \quad \text{with} \quad \mathcal{T} = [-2^{-2}, 2^{-1} - 2^{1-p}] \setminus \{0\}, \tag{6}$$

together with $\varphi \in \mathbb{R}$ and $\mu \in \mathbb{Z}$, such that

$$\log_b(2) = \varphi \cdot 2^\mu \quad \text{and} \quad 1 \le \varphi < 2. \tag{7}$$

Then computing $\log_b(m/2^\tau)$ in (5) may be done through the evaluation of $\ell(t)$ defined as:

$$\ell(t) = \frac{\log_b(1+t)}{2^\mu \cdot t} = \frac{\log_2(1+t)\cdot\varphi}{t}. \tag{8}$$

Note that for a given value b, the values φ and μ are constant and they are computed at implementation-time. Now since φ in (7) lies $[1,2)$, using the fact that $\log_2(1+t)/t \in (1,2)$, $\forall\ t \in \mathcal{T}$, we deduce that $\ell(t)$ remains in $(1,4)$. And in that case no loss of accuracy occurs. Therefore from (8), if we note $h = 2^\delta \cdot t$ with $|h| \ge 2^{-1}$, we have

$$\log_b(m/2^\tau) = h \cdot \ell(t) \cdot 2^{\mu-\delta}, \tag{9}$$

where $h \in \mathcal{H}$ with

$$\mathcal{H} = [-1+2^{3-p}, -2^{-1}] \cup [2^{-1}, 1-2^{2-p}], \tag{10}$$

and $|h\cdot\ell(t)| \in [2^{-1}, 4)$. And since $2^{-p} \le |t| \le 2^{-1} - 2^{1-p}$, we deduce that δ lies in $\{1, \cdots, p-1\}$.

Finally from (5), (7), and (9), $\log_b(x)$ is defined as follows:

$$\log_b(x) = \left(\varphi\cdot(e+\tau) + 2^{-\delta}\cdot h\cdot\ell(t)\right)\cdot 2^\mu. \tag{11}$$

C. How to get correctly rounded result?

When x is not a straightforward input, in order to determine the correctly rounded result from (11), let us define $u \in \mathbb{R}$ such that $|u| \in [1,2)$, and $c \in \mathbb{Z}$ as follows:

$$u \cdot 2^c = \left(\varphi\cdot(e+\tau) + 2^{-\delta}\cdot h\cdot\ell(t)\right). \tag{12}$$

Hence using (11) and (12):

$$\text{RN}(\log_b(x)) = (-1)^{\texttt{sign}(u)} \cdot \text{RN}(|u|)\cdot 2^{\mu+c}. \tag{13}$$

Since in our cases, $\log_b(x)$ does never underflow nor overflow, $\text{RN}(|u|)$ and $\mu + c$ can be seen as the significand and the exponent of the result, respectively, while its sign corresponds to the sign of u, and where the value of c falls in one of the following two cases:

- Case 1: $e+\tau = 0$. Here we have $|u|\cdot 2^c = 2^{-\delta}\cdot|h\cdot\ell(t)|$. Since $\delta \ge 1$ and $|h\cdot\ell(t)| \ge 2^{-1}$, we deduce that
$$c \in \{-1-\delta, \cdots, 0\}.$$
- Case 2: $e+\tau \neq 0$. From the definition of $\log_b(x)$ in (5), using (7) and (11), we can deduce that $|u|\cdot 2^c > |\log_2(0.75)|\cdot\varphi$. Since $\varphi > 1$, we have $c \ge -2$. Moreover when $|\varphi\cdot(e+\tau)| \ge 4$, $|u|\cdot 2^c \ge 1$ and the value c is roughly the size of $\varphi\cdot(e+\tau)$:
$$\lfloor \log_2(\varphi\cdot|e+\tau|)\rfloor - 1 \le c \le \lceil \log_2(\varphi\cdot|e+\tau|)\rceil + 1.$$
And if we denote by $c_{\max}$ the largest value of c, we have $c_{\max} = \lceil \log_2\left(\varphi\cdot\max(|e_{\min}-p+1|, |e_{\max}+3|)\right)\rceil + 1$.

Note that when $\text{RN}(|u|) = 2$, the significand and the exponent become 1 and $\mu+c+1$, respectively.

Now if we note $r = \text{RN}(|u|)$, since u cannot be computed exactly, we must approximate it by a finite-precision value $\hat{u}$. Once $\hat{u}$ is known accurately enough, the computation of r is straightforward and done as follows, where $\text{truncate}(\cdot)_{p-1}$ is the truncation function on $p-1$ fraction bits:

$$r = \text{truncate}(|\hat{u}| + 2^{-p})_{p-1}. \tag{14}$$

IV. Sufficient conditions to compute correct rounding

The problem is now reduced to the approximation of u in (12), by the computation of a value $\hat{u}$, such that

$$|u - \hat{u}| < \epsilon. \tag{15}$$

Here the value of the error bound ϵ depends on the function and is linked to the *Table Maker's Dilemma* [17]. This will be discussed further in Section VI-A. For the computation of $\hat{u}$, our implementation process follows 3 main steps, as what is done in [14] for roots and division.

1) First we consider the value u as the exact result of the function $F(t', h', e', \tau', \delta', c')$ defined as:
$$F = \left(\varphi\cdot(e'+\tau') + 2^{-\delta'}\cdot h'\cdot\frac{\log_b(1+t')}{2^\mu\cdot t'}\right)\cdot 2^{-c'},$$
at the point $(t', h', e', \tau', \delta', c') = (t, h, e, \tau, \delta, c)$.
2) Second, since F cannot be evaluated directly using arithmetic operations available on computers, we approximate this function F by a function P over $\mathcal{T}\times\mathcal{H}\times\mathcal{E}\times\{0,1\}\times\mathcal{D}\times\mathcal{C}$, with $\mathcal{T}$ and $\mathcal{H}$ in (6) and (10), respectively, and
$$\mathcal{E} = [e_{\min}, e_{\max}],\ \mathcal{D} = \{1, \cdots, p-1\},\ \text{and}\ \mathcal{C} = \{-p, \cdots, c_{\max}\}.$$
Here a good choice for $P(t', h', e', \tau', \delta', c')$ is:
$$P = \left(\varphi\cdot(e'+\tau') + 2^{-\delta'}\cdot h'\cdot a(t')\right)\cdot 2^{-c'},$$
where $a(t')$ is a polynomial approximant of the function $\log_b(1+t')/(2^\mu\cdot t')$ over $\mathcal{T}$.
3) Third we evaluate the function P at the point $(t, h, e, \tau, \delta, c)$ by a finite-precision evaluation program $\mathcal{P}$, and we assign the result to $\hat{u}$.

Most of the time, the constant φ in P is not exactly representable in finite precision. Hence the function to be evaluated is therefore the function $\hat{P}(t', h', e', \tau', \delta', c')$ defined by:

$$\hat{P} = \left(\hat{\varphi}\cdot(e'+\tau') + 2^{-\delta'}\cdot h'\cdot a(t')\right)\cdot 2^{-c'},$$

where

$$\hat{\varphi} = \text{RD}_{k-1}(\varphi) \quad \text{and} \quad 0 \le \varphi - \hat{\varphi} < 2^{2-k}. \tag{16}$$

Here the value of φ is rounded downward to ensure that it remains less than 2 in the implementations.

Recall that our goal is to automate the design of correctly rounded implementations. For this purpose, we need to exhibit some sufficient conditions on the accuracy of the different parts of the program, such as the targeted accuracy is achieved. Here and hereafter, we denote by $\alpha(a)$ the approximation error of $a(t)$:

$$\alpha(a) = \max_{t'\in\mathcal{T}}\left|\frac{\log_b(1+t')}{2^\mu\cdot t'} - a(t')\right|,$$

and by $\rho(\mathcal{P})$ the evaluation error of $\mathcal{P}$:

$$\rho(\mathcal{P}) = \max \left| \hat{P}(t', h', e', \tau', \delta', c') - \mathcal{P}(t', h', e', \tau', \delta', c') \right|$$

for all $(t', h', e', \tau', \delta', c') \in \mathcal{T} \times \mathcal{H} \times \mathcal{E} \times \{0, 1\} \times \mathcal{D} \times \mathcal{C}$.

Property 4. *Given u and $(t, h, e, \tau, \delta, c)$, together with the polynomial a, the functions P, $\hat{P}$, and the program $\mathcal{P}$, let us denote $\hat{u} = \mathcal{P}(t, h, e, \tau, \delta, c)$. If*

$$(2 - 2^{3-p}) \cdot \alpha(a) + \rho(\mathcal{P}) \leq \epsilon - 2^{6-k}, \tag{17}$$

then the value $\hat{u}$ satisfies (15)*.*

Proof: Using the definition of F, P, $\hat{P}$, and $\mathcal{P}$, it follows:

$$\begin{aligned} |u - \hat{u}| &= |F(t, h, e, \tau, \delta, c) - \mathcal{P}(t, h, e, \tau, \delta, c)| \\ &\leq |F(t, h, e, \tau, \delta, c) - P(t, h, e, \tau, \delta, c)| + \\ &\quad \left| P(t, h, e, \tau, \delta, c) - \hat{P}(t, h, e, \tau, \delta, c) \right| + \\ &\quad \left| \hat{P}(t, h, e, \tau, \delta, c) - \mathcal{P}(t, h, e, \tau, \delta, c) \right| \\ &\leq 2^{-\delta - c} \cdot |h| \cdot \left| \frac{\log_b(1+t)}{2^\mu \cdot t} - a(t) \right| + \\ &\quad 2^{-c} \cdot |e + \tau| \cdot |\varphi - \hat{\varphi}| + \rho(\mathcal{P}). \end{aligned}$$

Since $|h| < 1 - 2^{2-p}$, using the definition of δ and c in Section III, and of the floor function $\lfloor \cdot \rfloor$, it follows from (16), that:

$$|u - \hat{u}| < (2 - 2^{3-p}) \cdot \alpha(a) + 2^{6-k} + \rho(\mathcal{P}),$$

which concludes the proof. ∎

From Property 4, we know that if (17) holds, then the computed value $\hat{u}$ is accurate enough to guarantee the correct rounding. Note that when $b = 2$, $|\varphi - \hat{\varphi}| = 0$, and thus this condition becomes $(2-2^{3-p}) \cdot \alpha(a) + \rho(\mathcal{P}) \leq \epsilon$. Therefore, the goal is now to build a polynomial $a(t')$ together with a program $\mathcal{P}$ so as (17) holds. The first step consists in computing the finite-precision polynomial $a(t')$. The evaluation error $\rho(\mathcal{P})$ being non-negative, the polynomial a must be such as:

$$\alpha(a) < (\epsilon - 2^{6-k})/(2 - 2^{3-p}). \tag{18}$$

Typically for the binary32 format, if we target an accuracy $\epsilon = 2^{-51}$, we need $k = 64$. And in this case, $\alpha(a)$ must be less than $\approx 2^{-51.99}$. Then once the polynomial a is computed, having

$$\alpha(a) \leq \theta \quad \text{where} \quad \theta < (\epsilon - 2^{6-k})/(2 - 2^{3-p}),$$

the objective is to find a program $\mathcal{P}$ such that $\rho(\mathcal{P}) \leq \eta$, where

$$\eta < \epsilon - 2^{6-k} - (2 - 2^{3-p}) \cdot \theta. \tag{19}$$

The bound η depends on the underlying arithmetic and the error entailed by the evaluation of a. This is discussed further in Section V-C.

V. Automated implementation

Given b as in (3), the precision p, and the exponent $e_{\max}$, this section gives some guidelines to implement $\log_b(x)$ on integer processors, by using the algorithm presented in this article. This results in a code that takes as input and returns a floating-point datum encoded by an integer using the binary interchange format encoding described in [1, § 2]. Note that these guidelines being parametrized by $(b, p, e_{\max})$, a tool can be derived to automate this implementation process.

A. Assumption

We assume here that the floating-point input and output are encoded using k'-bit integers, and all the internal computations are done in k-bit two's complement arithmetic, with $k \geq k'$ and $k', k > 0$. For the binary32 format, we have $(k', k) = (32, 64)$.

Moreover for the implementation purpose, in addition to the usual integer operations available in the standard C language, we consider the following routines available, where X and Y are k-bit integers:

- `mul`$(X, Y) = \lfloor X \cdot Y \cdot 2^{-k} \rfloor$,
- `min`(X, Y) : smaller value between X and Y,
- and `nlz`(X) : number of leading zeros of X.

Finally, from Properties 2 and 3 we know that no underflow nor overflow occurs in our test cases. This simplifies the implementations, since no special handling has to be designed.

B. Guidelines

The special and straightforward input handling as well as the input unpacking and output packing are implemented in a similar way to what is done in [14], and it is not detailed in this article. In our context, input unpacking results in:

- a k-bit unsigned integer $F = (m - 1) \cdot 2^k$,
- and a k'-bit signed integer E representing e.

The goal is now to write some pieces of code to help in computing the approximation $\hat{u}$. Particularly, we compute an integer U such that $\hat{u} = U \cdot 2^n$ with n a well-chosen exponent. For doing this, let us define the following k-bit signed integers:

- $L = \hat{\varphi} \cdot 2^{k-2}$, such that $1 \leq L < 2$,
- $E_i = (E + \tau) \cdot 2^{k-i}$ where i is the number of useful bits in the k-bit integer representation of $e + \tau$, that is, excluding redundant sign bits,
- $M = m \cdot 2^{k-2}$, $T = t \cdot 2^k$, $H = h \cdot 2^{k-1}$,
- and $V =$ `poly_eval`(T, H) encoding the evaluation result of $h \cdot a(t)$, where `poly_eval` is a program that works on k-bit integers.

Here, instead of evaluating $a(t)$ and multiplying the result by h, we tend to distribute the multiplication by h inside the evaluation of $a(t)$. And since $|h \cdot \ell(t)| < 4$, this program can always be built so as to return a signed integer

$$V = \big(h \cdot a(t)\big) \cdot 2^{k-3}.$$

Computation of M, T, δ, and H. The integer M is computed from F that encodes the fractional part of the input significand, as follows:

$$\begin{aligned} M &= m \cdot 2^{k-2} = (1 + f) \cdot 2^{k-2} \\ &= 2^{-2} \cdot F + 2^{k-2}. \end{aligned}$$

To compute T, we have to compute first the value of τ, which is actually defined as the result of the test $[m \geq 1.5]$, that is, $[M \geq 1.5 \cdot 2^{k-2}]$. Finally, the integer T is

$$\begin{aligned} T &= t \cdot 2^k = (m/2^\tau - 1) \cdot 2^k \\ &= (M/2^\tau - 2^{k-2}) \cdot 2^2, \end{aligned}$$

where $\tau \in \{0, 1\}$. Here is a piece of code to compute M, τ, and T.

```
M = (F >> 2) + C1;                    // C1 = 2^(k-2)
tau = (M >= C2);                 // C2 = 1.5 * 2^(k-2)
T = ((M >> tau) - C1) << 2;
```

Then to compute H, the key point is to determine δ. It corresponds actually to the number of sign bits in the bitstring of T. Indeed, if $T = 2^{k-2}$, then $t = 0.25$ and $\delta = 1$. Hence to do so, all the bits of T are first inverted *only* when it is negative: this is done by combining a XOR and a right shift operation as shown at Line 1 in the code below, and where the shift is a *signed shift* involving *sign extension* [18, § 5.3]. Then a `nlz` instruction is used to count the number of leading zeros appearing in the resulting integer, which corresponds to δ.

```
1 inv_T = T ^ (T >> C3);                    // C3 = k-1
2 delta = nlz(inv_T);
3 H = T << delta;
```

Note that the C standard requires that a shift operation of a k-bit integer is done by $\{0, \cdots, k-1\}$ bits to be valid [19]. Hence at Line 3, since $\delta \in \{1, \cdots, p-1\}$ in (9) and $p < k$, the shift operation is well-defined.

Computation of U. From T and H, we are now able to compute U encoding $\hat{u}$. It is done through the computation of two k-bit integers U_1 and U_2, representing $\varphi \cdot (e + \tau)$ and $2^{-\delta} \cdot h \cdot a(t)$, respectively.

Let us start by U_1: Since our multiplication is a *truncated multiplication*, instead of approximating $L \cdot E$, we prefer to approximate $L \cdot E_i$ in order to maintain a sufficient accuracy on the truncated result. Hence the first step consists in computing the integer $E_i = (E+\tau) \cdot 2^{k-i}$. The parameter i is determined similarly to δ: it equals to the wordlength of the integer $E+\tau$ (that is k') minus the number of useless redundant sign bits:

$$i \quad = \quad k' - (\texttt{nlz(inv\_E)} - 1).$$

And the integer E_i is obtained by left shifting $E+\tau$ by $k-i$ bits, as shown the piece of code below.

```
inv_E = (E + tau)^((E + tau) >> C4);  // C4 = k'-1
i = C5 - nlz(inv_e);                  // C5 = k'+1
Ei = (E + tau) << (C6 - i);              // C6 = k
```

This results in the following code, where U_1 is defined as:

$$\begin{aligned} U_1 \quad &= \quad \texttt{mul}(L, E_i) \; = \; \lfloor L \cdot E_i \cdot 2^{-k} \rfloor \\ &= \quad \lfloor \hat{\varphi} \cdot (e+\tau) \cdot 2^{k-i-2} \rfloor. \end{aligned}$$

```
U1 = mul(L, Ei);
```

Note that when $b = 2$, we have $\varphi = 1$ and $L = 2^{k-2}$. Then we simply have $U_1 = E_i \cdot 2^{-2}$. In this case, the above piece of code is simplified as follows.

```
U1 = Ei >> 2;
```

Now let us continue with U_2: The multiplication by $2^{-\delta}$ in $2^{-\delta} \cdot h \cdot a(t)$ is implicit, and it is not implemented. Hence we have $U_2 = V$, and

$$U_2 = \left(2^{-\delta} \cdot h \cdot a(t)\right) \cdot 2^{k-3+\delta}.$$

Remark that in the particular case $t = 0$, we have $T = 0$ and $H = 0$. And whatever the polynomial coefficients, we have $V = 0$. Hence we do not need to handle the case $t = 0$ separately, and it can be handled in the general flow.

Once U_1 and U_2 are known, they must be scaled to the same power of 2 to be added. This scaling operation must be carried out such as no overflow occurs on any of both integers U_1 and U_2, and in order to keep at most accuracy. Since $i \geq 1$ and $\delta \geq 1$, we know that $i + 2 \geq 3$ and $3 - \delta \leq 2$, and thus $k - i - 2 \leq k - 3 + \delta$. In this case, U_2 must be scaled to fit in the format of U_1. Hence we have:

$$U_2 \quad = \quad \lfloor U_2 / 2^{i+\delta-1} \rfloor,$$

where $i+\delta-1 \geq 0$. However, when $e+\tau = 0$ then $E+\tau = 0$, and $U_1 = 0$: in this case, in order not to loss accuracy, U_2 must not be scaled. And in order to avoid branches in the resulting code, we may use a mask that cancels the scaling of U_2 when $E + \tau = 0$. The piece of code below shows this scaling step, where `mask` is a k'-bit integer whose bitstring is only composed of 0 when $E+\tau = 0$ and of 1 otherwise. Then, by using a AND operation, the shift right operand is either 0 (when $E + \tau = 0$) or the value we want U_2 to be shifted.

```
mask = 0 - ((E + tau) != 0);
U2 = U2 >> (min(k-1, i + delta - 1) & mask);
```

In this code, a `min` instruction is used to ensure that the shift remains valid. If $\min(k-1, i+\delta-1) = k-1$ and $E + \tau \neq 0$, the resulting integer U_2 is 0 as if we would have right shifted U_2 of $i+\delta-1$ positions. But here the value of i is at most

$$i_{\max} = \lceil \log_2(|e_{\min} - p + 1|) \rceil + 1 + [e_{\min} - p + 1 = -2^{\text{exponent}}],$$

since $|e_{\min} - p + 1| \geq 2$ and $e_{\min} = 1 - e_{\max}$. And using $\delta \leq p-1$, it follows that if $i_{\max} < k - p + 2$ then the `min` instruction above is not required, which simplified the code as below.

```
U2 = U2 >> ((i + delta - 1) & mask);
```

Finally, U is computed by adding U_1 and U_2. Notice that when $E + \tau \neq 0$, we have $i \geq 2$. In this case by construction, both $|U_1|$ and $|U_2|$ are less than 2^{k-2}. Hence we conclude that no overflow occurs when computing U.

```
U = U1 + U2;
```

Computation of c. The quantity U represents the approximation $\hat{u}$ such that $U = \hat{u} \cdot 2^n$. It follows from (12) and the definition of U that

$$n = \begin{cases} c + k - i - 2, & \text{if } e + \tau \neq 0, \\ c + k - 3 + \delta, & \text{if } e + \tau = 0. \end{cases}$$

Since $|U| \neq 0$, we have $1 \leq |U| / 2^{\texttt{nlz}(|U|)-1} < 2$ and $k = \texttt{nlz}(|U|) + 1 + n$. Hence c is defined as:

$$c = \begin{cases} i + 1 - \texttt{nlz}(|U|), & \text{if } e + \tau \neq 0, \\ 2 - \delta - \texttt{nlz}(|U|), & \text{if } e + \tau = 0. \end{cases}$$

To avoid branches in the resulting code, when $e + \tau \neq 0$, we may define $c = 2 - \delta - \texttt{nlz}(|U|) - (1 - i - \delta)$, where subtracting $1 - i - \delta$ only when $e + \tau = 0$ is done using a mask as previously.

It follows that the computation of c requires the determination of $|U|$. From [18, § 2.4], we know that $|U|$ may be

computed similarly to δ or i above: by inverting all the bits of U and adding 1 if this one is negative. The code below shows the computation of $|U|$, c, and n.

```
absU = (U ^ (U >> C3) - (U >> C3);
c = 2 - delta - nlz(absU) - ((1-i-delta) & mask);
n = k - 1 - nlz(absU);
```

Note that by construction, `nlz`($|U_1|$) = 2. It follows that `nlz`($|U|$) $\in \{1, 2, 3\}$. Hence we deduce that

$$c \in \{i, i-1, i-2\} \quad \text{when} \quad e + \tau \neq 0. \tag{20}$$

This point will be useful in Section V-C when dealing with error analysis.

Computation of the result sign, exponent, and significand. For the packing process, we have to compute the sign of the result, together with its biased exponent and significand. First the sign of the result $S \in \{0, 1\}$ can be read on the most significant bit of the integer U.

```
S = (U >> C3) & 1;
```

Second let us compute the significand r of the result. For doing this, let R be a k'-bit integer such as $R = r \cdot 2^{p-1}$. Using (14), together with the definition of U and $\hat{u}$, R can be computed as follows:

$$R = \lfloor (|U| + 2^{n-p})/2^{n-p+1} \rfloor.$$

When X is a k-bit integer and $x \in \{0, \cdots, k-1\}$, $\lfloor X/2^x \rfloor$ can be implemented using a right shift operation. Hence, since $0 \leq n - p \leq k - 1$, the following piece of code computes R.

```
R = (absU + (1 << (n-p))) >> (n-p+1);
```

Third from (13), we know that the exponent d of the result is computed as follows:

$$d = \begin{cases} \mu + c, & \text{if RN}(|u|) \neq 2, \\ \mu + c + 1, & \text{if RN}(|u|) = 2. \end{cases}$$

As shown in [14], the second case may be ignored, and the exponent update is done within the packing process. Particularly the idea is to compute D, a k'-bit exponent representing the biased value of $\mu + c - 1$:

$$D = \mu + c - 1 + e_{\max} = \mu + c - e_{\min}.$$

Recall the μ is known at implementation-time, and it depends on the value b.

```
D = c + C7;                                  // C7 = mu - emin
```

C. Error analysis

The goal is here to bound the error entailed by the evaluation of the function $\hat{P}$ in finite-precision arithmetic by the program $\mathcal{P}$, that is, to bound the quantity $|\hat{P} - \hat{u}|$, by η as in (19), and to determine a sufficient condition on $\rho(a)$ defined by:

$$\rho(a) = \max_{(t',h') \in \mathcal{T} \times \mathcal{H}} \left| h' \cdot a(t') - V \cdot 2^{3-k} \right| \tag{21}$$

such as the error bound η in (19) is satisfied. For this purpose, let us distinguish two cases:

- Case 1: $e + \tau = 0$. In this case, the error comes only from the computation of U_2. By definition, we have:

  $$\left| \hat{P} - \hat{u} \right| = \left| h \cdot a(t) - U_2 \cdot 2^{3-k} \right| \cdot 2^{-c-\delta}.$$

  Since $c \geq -1 - \delta$, using (21) we deduce

  $$\left| \hat{u} - \hat{P} \right| \leq 2 \cdot \rho(a). \tag{22}$$

- Case 2: $e + \tau \neq 0$. By definition, we have:

  $$\begin{aligned} \left| \hat{P} - \hat{u} \right| \quad \leq \quad & \left| \hat{\varphi} \cdot (e + \tau) - U_1 \cdot 2^{i+2-k} \right| \cdot 2^{-c} + \\ & \left| 2^{-\delta} \cdot h \cdot a(t) - U_2 \cdot 2^{i+2-k} \right| \cdot 2^{-c}. \end{aligned}$$

  The `mul` instruction returning the truncated result of a finite-precision multiplication, since $c \geq -2$ and $i \geq 1$, and using (20) and (21) we deduce

  $$\left| \hat{P} - \hat{u} \right| < 4 \cdot \rho(a) + 2^{5-k}. \tag{23}$$

Property 5. *Given the function $\hat{P}$, a program $\mathcal{P}$, and $(t, h, e, \tau, \delta, c)$ together with $\hat{u} = \mathcal{P}(t, h, e, \tau, \delta, c)$. If*

$$\rho(a) < (\epsilon - 2^{6-k} - (2 - 2^{3-p}) \cdot \theta - 2^{5-k})/4$$

then the bound η satisfies (19)*.*

Proof: Using (22) and (23), it follows that η in (19) is

$$\eta = 4 \cdot \rho(a) + 2^{5-k},$$

which concludes the proof. ∎

Finally, using Property 5, the remaining part consists in designing an evaluation program for a, so that the evaluation $\rho(a)$ satisfies $\rho(a) \leq \kappa$, where

$$\kappa < (\epsilon - 2^{6-k} - (2 - 2^{3-p}) \cdot \theta - 2^{5-k})/4. \tag{24}$$

VI. Experiments

This section gives some results concerning the automated implementation of $\log_b(x)$ for the binary32 and binary64 floating-point formats.

A. Required error bound to ensure correct rounding

The first step of the implementation process consists in determining the error bound ϵ in (15) required to ensure correct rounding: this problem is known as the *Table's Maker Dilemma*. (See [16, § 12] for details.)

To our knowledge, for the logarithm functions, this bound has been published for the binary64 but not for the binary32 floating-point format, except in [13] for the case $b = 2$. Hence to determine this bound for the other cases, we designed an approach based on exhaustive testing. This consists in computing the smallest distance between $\log_b(x')$ and a breakpoint in precision p, for all the floating-point inputs $x' \in [\alpha, \Omega]$. Since $\log_b(x')$ cannot be computed exactly, the idea is (1) to compute an approximation v of $\log_b(x')$ in a precision $p' \gg p$, (2) to round v to the closest breakpoint in precision p and to assign the result to $\hat{v}$, and (3) to compute the distance $d = |v - \hat{v}|$. Using Sterbenz lemma, in precision p', the distance d can be computed exactly. Finally, the distance d being affected by the error on v, it is increased by $2^{-p'}$. This process is illustrated

b	2	exp(1)	10
accuracy (# of bits)	51	58	56
smallest degree of a	19	21	21

Table II: Accuracy and smallest polynomial degree needed for correctly rounded $\log_b(x)$ in binary32.

in Figure 1. Table II shows the results we obtained with our approach implemented using the MPFR library,[1] and where $(p, p') = (24, 80)$. For example, to implement $\log_2(x)$, we need $\hat{u}$ in (15) so that $|u - \hat{u}| < 2^{-51}$.

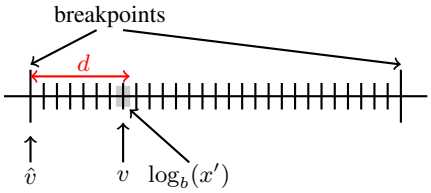

Figure 1: Approach to determine the required error bound ϵ.

B. Implementation details for the binary32 format

Using Table II together with (18) and (24), and the guidelines of Section V-B, we are now able to build the whole implementation of various logarithm functions. In this section, we target the binary32 format.

1) Determination of the wordlength k: Since $\alpha(a)$ in (18) and $\rho(a)$ in (24) are non-negative, k must be chosen so as $\epsilon - 2^{6-k} - 2^{5-k} > 0$. Using Table II, we can conclude that $k = 64$ is suitable for these implementations, except for $\log(x)$, where this bound becomes negative. In this case $k = 96$ or $k = 128$ would be preferred. However, we can observe that for $\log(x)$ only one input x requires a 58-bit accuracy: $x = 127837836949849943048192$, and one input require a 57-bit accuracy: $x = 58037908$. For all the inputs but these ones, an accuracy of 56 bits if sufficient to enable to decide correct rounding. These inputs can thus be ignored during the implementation process. At the end, we only have to test whether correct rounding is achieved also for these: If it is not the case, we just have to consider these as *straightforward inputs* and to test them separately in the final code. Hence, for the rest of this section, we consider $k = 64$, and a required accuracy of 56 bits for $\log(x)$.

2) Polynomial approximation and evaluation: Let us consider the $\log_{10}(x)$ function. It follows from Table II and (18) that $\epsilon < 2^{-56}$. Hence the approximation error of the polynomial approximant a must satisfy $\alpha(a) \leq \theta$ where

$$\theta < (2^{-56} - 2^{-58})/(2 - 2^{-21}) \approx 2^{-57.41}. \qquad (25)$$

Such a polynomial is built using the Sollya tool,[2] as follows:

(1) Using the `guessdegree` routine, we determine the smallest degree d_a of a so as the bound in (25) can be reached.
(2) Then we compute a degree-d_a polynomial approximant a of $\log_{10}(1+t)/(2^{-2} \cdot t)$ over $\mathcal{T}$ using the `fpminimax` routine, which returns the best polynomial with coefficients in a given format. As shown in Table II, here a is a degree-21 polynomial, where each coefficient a_i is encoded on k bits, that is, using a 64-bit signed integer A_i. In order to keep at most accuracy on each coefficient, each A_i satisfy $A_i = a_i \cdot 2^{f_i}$, where f_i is chosen so as $2^{k-2} \leq |A_i| < 2^{k-1}$, that is, $2^{62} \leq |A_i| < 2^{63}$.
(3) Finally, we determine the certified bound θ using the `supnorm` routine. In this example, the approximation error bound $\theta \approx 2^{-58.62}$,[3] which satisfies (25).

Once the bound θ is known, the key point is now to build a code to evaluate $a(t)$, whose evaluation error satisfies the bound (24). It consists in writing a code so as $\rho(a) \leq \kappa$ where

$$\kappa < \left(2^{-56} - 2^{-58} - (2 - 2^{-21}) \cdot \theta - 2^{-59}\right)/4.$$

This is done by calling the CGPE tool.[4] Given the polynomial coefficients and a bound on the evaluation error, it enables to synthesize codes for evaluating the polynomial in fixed-point arithmetic, that is, using only integer arithmetic, and to guarantee that its evaluation error is within the given bound. This accuracy certification step is performed with the help of the Gappa tool, that uses interval arithmetic combined with rewriting rules in order to formally prove numerical properties on programs.[5] If such a program cannot be found, we increase the degree of the polynomial and restart the generation process. Table II shows the smallest polynomial degree for each function, while for $\log(x)$, a degree-22 polynomial is actually used in the implementation, since no accurate enough program could be found with a degree-21 polynomial.

Function	Nb. of instr.	Latency	IPC	Scheme
$\log_2(x)$	883	251	3.51	Horner4
$\log(x)$	869	313	2.77	Horner
$\log_{10}(x)$	837	299	2.79	Horner

Table III: Number of instructions, latency (cycles), and IPC for correctly-rounded binary32 $\log_b(x)$ on the ST231.

C. Performance on the ST231

Using the process above, we have implemented in a few minutes the three functions $\log(x)$, $\log_2(x)$, and $\log_{10}(x)$, for the binary32 format. We have compiled the generated codes on the ST231, a 4-issue 32-bit VLIW integer processor, in O3 optimization level. To evaluate the underlying polynomials, we tried four usual rules: Horner, 2nd-order Horner, 4th-order Horner, and Estrin [23, § 4.6.4]. For $\log(x)$ and $\log_{10}(x)$, we are able to certify the evaluation error bound only with Horner rule. Our first observation shows that Horner rule does not expose enough ILP (Instruction-Level Parallelism) to make the most of the ST231. Conversely, evaluating such high degree polynomials in 64-bit arithmetic with Estrin rule exposes too much ILP to be compiled on this architecture without register spilling. We conclude that 4th-order Horner rule is the most suitable schemes in our context, leading to a gain of $\approx 7\%$ in latency compared to Horner for the $\log_2(x)$ function. Table III shows the performance of these three codes on the ST231, where `mul` and `nlz` instructions are emulated in software [18].

[1] See http://www.mpfr.org.
[2] See http://sollya.gforge.inria.fr/ and [20].
[3] $\theta = 288796683762110203271340989018514898310838 55 \cdot 2^{-203}$.
[4] See http://cgpe.gforge.inria.fr/ and [21].
[5] See http://gappa.gforge.inria.fr/ and [22].

	binary32		binary64	
Function	Latency	IPC	Latency	IPC
$\log_2(x)$	51	2.54	264	3.08
$\log(x)$	64	1.98	269	3.59
$\log_{10}(x)$	58	2.05	269	3.56

Table IV: Latency (cycles) and IPC of faithfully rounded binary32 and binary64 $\log_b(x)$ on the ST231.

We observe that using our automated approach leads to a correctly rounded implementation in 251 to 300 cycles for all these functions. In [24], a 47-cycle binary32 implementation is given for $\log_2(x)$: but it works an 32-bit arithmetic only, and no guarantee is given on the accuracy of the output. Finally, in our implementations, we reach an IPC (Instructions Per Cycle) of ≈ 3.5, while using Horner rule would lead to an IPC of ≈ 2.8. This shows clearly the interest of our approach, since no more than 4 instructions could be launched at each cycle on the ST231.

D. Extension to faithful rounding

All along this article, our goal was to reach correctly rounded implementations. If we now relax this constraint, and we target faithful rounding: in this case, the bound in (15), is no longer the ones in Table II, but we only need $\epsilon = 2^{-p}$, where p is the precision of the considered format. Table IV shows the performance on the ST231 of faithfully rounded implementations of these three functions. Here we use $k = 64$ and degree-20 polynomials evaluated with 4th-order Horner for binary64 implementations, while $k = 32$ and degree-9 polynomials are enough except for $\log(x)$ which required a degree-10 polynomial, and where these are evaluated with 2nd-order Horner rule, but for $\log_2(x)$ which relies on Estrin.

We observe that our approach enables to write faithfully rounded implementations with a latency of 51 up to 64 cycles in binary32 and 264 up to 269 cycles in binary64, with a relatively high IPC. This shows that computing correct rounding may be costly on this architecture, about 200 cycles for the binary32 format. This mainly comes from the degree of the polynomial, which is smaller in this case, and from the fact that 32-bit arithmetic is enough. Hence this reinforces the interest in providing faithful implementations, for applications and context where correct rounding is not necessary.

VII. Conclusion and future works

This article addresses the automated design of logarithm function implementations. We show the interest of our approach for the correctly rounded implementation of binary32 $\log(x)$, $\log_2(x)$, and $\log_{10}(x)$, resulting in full implementations in 251 up to 300 cycles on the ST231 processor. We also show how this parametrized approach can be used to write faithfully rounded binary32 and binary64 implementations, reducing significantly the implementation latency. Note that the work presented here for $b \in \{2, \exp(1), 10\}$ can be extended to implement any $\log_b(x)$ functions.

The research direction is threefold: First this could be interesting to extend this approach to other transcendental functions like $\exp_b(x)$, and to observe what performances could be achieved on integer processors. Second all our experiments have been carried out on integer processors. It could be interesting to evaluate our implementations on other architectures like general purpose processors, and to observe how they compare with the other methods of the literature. Third in our experiments, we used CGPE as a backend for polynomial evaluations, where all the polynomials are evaluated using classical schemes. Since CGPE enables to look for a *fast* evaluation scheme, this could be interesting to study the impact on performance and accuracy of this evaluation scheme.

References

[1] "IEEE standard for floating-point arithmetic," IEEE Std. 754-2008, pages 1–58, Aug. 2008.

[2] P. T. P. Tang, "Table-driven implementation of the logarithm function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 16, no. 4, pp. 378 – 400, Dec. 1990.

[3] P. Markstein, *IA-64 and Elementary Functions : Speed and Precision*, ser. Hewlett-Packard Professional Books. Prentice Hall, 2000.

[4] C. S. Anderson, S. Story, and N. Astafiev, "Accurate math functions on the Intel IA-32 architecture: A performance-driven design," in *7th Conference on Real Numbers and Computers*, 2006, pp. 93–105.

[5] S. Gal, "Computing elementary functions: A new approach for achieving high accuracy and good performance," in *Proceedings of the Symposium on Accurate Scientific Computations*. London, UK, UK: Springer-Verlag, 1986, pp. 1–16.

[6] S. Gal and B. Bachelis, "An accurate elementary mathematical library for the IEEE floating point standard," *ACM Transactions on Mathematical Software*, vol. 17, no. 1, pp. 26–45, Mar. 1991.

[7] W. F. Wong and E. Goto, "Fast evaluation of the elementary functions in double precision," in *Twenty-Seventh Annual Hawaii International Conference on System Sciences*, 1994, pp. 349–358.

[8] W. F. Wong and E. Goto, "Fast evaluation of the elementary functions in single precision," *IEEE Transactions on Computers*, vol. 44, no. 3, pp. 453–457, Mar. 1995.

[9] J. Detrey and F. de Dinechin, "A parameterizable floating-point logarithm operator for FPGAs," in *39th Asilomar Conference on Signals, Systems & Computers*. IEEE, 2005.

[10] J. Detrey and F. de Dinechin, "Parameterized floating-point logarithm and exponential functions for FPGAs," *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing*, vol. 31, no. 8, pp. 537–545, 2007.

[11] W. F. Wong and E. Goto, "Fast hardware-based algorithms for elementary function computations using rectangular multipliers," *IEEE Transactions on Computers*, vol. 43, no. 3, pp. 278–294, Mar. 1994.

[12] F. de Dinechin, C. Lauter, and J.-M. Muller, "Fast and correctly rounded logarithms in double-precision," *RAIRO - Theoretical Informatics and Applications*, vol. 41, no. 1, pp. 85–102, 4 2007.

[13] M. Schulte and E. Swartzlander, "Exact rounding of certain elementary functions," in *Proc. of the 11th IEEE Symposium on Computer Arithmetic (ARITH'11)*, 1993, pp. 138–145.

[14] G. Revy, "Implementation of binary floating-point arithmetic on embedded integer processors - polynomial evaluation-based algorithms and certified code generation," Ph.D. dissertation, Univ. de Lyon - ENS de Lyon, 2009.

[15] *ST231 core and instruction set architecture – Reference manual*, 2008.

[16] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[17] V. Lefèvre, J.-M. Muller, and A. Tisserand, "The table maker's dilemma," Ecole normale supérieure de Lyon, Lyon, Tech. Rep., 1998.

[18] H. S. W. Jr., *Hacker's Delight*. Addison-Wesley, 2003.

[19] International Organization for Standardization, *Programming Languages – C*. Geneva, Switzerland: ISO/IEC Standard 9899:1999, Dec. 1999.

[20] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya: An environment for the development of numerical codes," in *Mathematical Software - ICMS 2010*, vol. 6327, September 2010, pp. 28–31.

[21] C. Mouilleron and G. Revy, "Automatic Generation of Fast and Certified Code for Polynomial Evaluation," in *Proc. of the 20th IEEE Symposium on Computer Arithmetic (ARITH'20)*, 2011, pp. 233–242.

[22] G. Melquiond, "De l'arithmétique d'intervalles à la certification de programmes," Ph.D. dissertation, ÉNS Lyon, France, 2006.

[23] D. E. Knuth, *Seminumerical Algorithms*, Third ed., ser. The Art of Computer Programming. Addison-Wesley, 1998, vol. 2.

[24] S.-K. Raina, "FLIP: a floating-point library for integer processors," Ph.D. dissertation, ÉNS Lyon, France, 2006.