Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

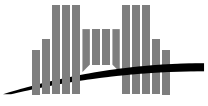
Fast and correctly rounded logarithms in double-precision

Florent de Dinechin, Christoph Lauter, Jean-Michel Muller

Septembre 2005

Research Report N° RR2005-37

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`



Fast and correctly rounded logarithms in double-precision

Florent de Dinechin, Christoph Lauter, Jean-Michel Muller

Septembre 2005

Abstract

This article is a case study in the implementation of a portable, proven and efficient correctly rounded elementary function in double-precision. We describe the methodology used to achieve these goals in the `crlibm` library. There are two novel aspects to this approach. The first is the proof framework, and in general the techniques used to balance performance and provability. The second is the introduction of processor-specific optimizations to get performance equivalent to the best current mathematical libraries, while trying to minimize the proof work. The implementation of the natural logarithm is detailed to illustrate these questions.

Keywords: floating-point, elementary functions, logarithm, correct rounding

Résumé

Cet article montre comment implémenter une fonction élémentaire efficace avec arrondi correct prouvé en double-précision. La méthodologie employée dans ce but par la bibliothèque `crlibm` présente deux aspects novateurs. Le premier concerne la preuve de l'arrondi correct, et plus généralement les techniques employées pour gérer les compromis entre performance et facilité de preuve. Le second est l'utilisation d'optimisations utilisant des caractéristiques les plus avancées des processeurs, ce qui permet d'obtenir une performance équivalente aux meilleures implémentations existantes. L'implémentation du logarithme népérien est décrite en détail à titre d'illustration.

Mots-clés: virgule flottante, fonctions élémentaires, logarithme, arrondi correct

1 Introduction

1.1 Correct rounding and floating-point elementary functions

Floating-point is the most used machine representation of the real numbers, and is being used in many applications, from scientific or financial computations to games. The basic building blocks of floating-point code are the operators $+, -, \times, \div$ and $\sqrt{}$ which are implemented in hardware (or with specific hardware assistance) on most workstation processors. Embedded processors usually require less floating-point performance and have tighter power constraints, and may therefore provide only software floating point emulation. On top of these basic operators, other building blocks are usually provided by the operating system or specific libraries: elementary functions (exponential and logarithm, trigonometric functions, etc.), operators on complex numbers, linear algebra, etc.

The IEEE-754 standard for floating-point arithmetic[2] defines the usual floating-point formats (single and double precision) and precisely specifies the behavior of the basic operators $+, -, \times, \div$ and $\sqrt{}$. The standard defines four rounding modes (to the nearest, towards $+\infty$, towards $-\infty$ and towards 0) and demands that these operators return the correctly rounded result according to the selected rounding mode. Its adoption and widespread use have increased the numerical quality of, and confidence in floating-point code. In particular, it has improved *portability* of such code and allowed construction of *proofs* of numerical behavior[17]. Directed rounding modes (towards $+\infty$, $-\infty$ and 0) are also the key to enable efficient *interval arithmetic*[26, 20].

However, the IEEE-754 standard specifies nothing about elementary functions, which limits these advances to code excluding such functions. Currently, several options exist: on one hand, one can use today's mathematical libraries, which are efficient but without any warranty on the accuracy of the results. These implementations use combinations of large tables [15, 16, 29] and polynomial approximations (see the books by Muller[28] or Markstein[25]). Most modern libraries are *accurate-faithful*: trying to round to nearest, they return a number that is one of the two FP numbers surrounding the exact mathematical result, and indeed return the correctly rounded result most of the time. This behavior is sometimes described using phrases like *99% correct rounding* or *0.501 ulp accuracy*. However, it is not enough when strict portability is needed, as was recently the case for the LHC@Home project: This project distributes a very large computation on a wide network of computers, and requires strict floating-point determinism when checking the consistency of this distribution, due to the chaotic nature of the phenomenon being simulated. Default libraries on different systems would sometimes return slightly different results.

When such stricter guarantees are needed, some multiple-precision packages like MPFR [27] offer correct rounding in all rounding modes, but are several orders of magnitude slower than the usual mathematical libraries for the same precision. Finally, there are currently three attempts to develop a correctly-rounded `libm`. The first was IBM's `libultim`[24] which is both portable and fast, if bulky, but lacks directed rounding modes needed for interval arithmetic. This project is no longer supported by IBM, but derivatives of the source code are now part of the GNU C library `glibc`. The second is `crlibm` by the Arénaire team at ENS-Lyon, first distributed in 2003. The third is Sun correctly-rounded mathematical library called `libmcr`, whose first beta version appeared in late 2004. Although very different, these libraries should return exactly the same values for all possible inputs, an improvement on current default situation.

This article deals with the implementation of a fast, proven correctly rounded elementary functions. The method used to provide efficient correct rounding has been described by Abraham Ziv [31], and is reminded in the sequel. The present article improves Ziv's work in two important aspects: First, it proves the correct rounding property. Second, the performance is greatly improved, especially in terms of worst-case execution time and memory consumption. These improvements are illustrated by a detailed description of the logarithm function.

1.2 The Table Maker's Dilemma and Ziv's onion peeling strategy

With a few exceptions, the image $\hat{y}$ of a floating-point number x by a transcendental function f is a transcendental number, and can therefore not be represented exactly in standard number systems. The correctly rounded result (to the nearest, towards $+\infty$ or towards $-\infty$) is the floating-point number that is closest to $\hat{y}$ (or immediately above or immediately below respectively).

A computer may evaluate an approximation y to the real number $\hat{y}$ with relative accuracy $\overline{\varepsilon}$. This means that the real value $\hat{y}$ belongs to the interval $[y(1-\overline{\varepsilon}), y(1+\overline{\varepsilon})]$. Sometimes however, this information is not enough to decide correct rounding. For example, if $[y(1-\overline{\varepsilon}), y(1+\overline{\varepsilon})]$ contains the middle of two consecutive floating-point numbers, it is impossible to decide which of these two numbers is the correctly rounded to the nearest of $\hat{y}$. This is known as the Table Maker's Dilemma (TMD) [28].

Ziv's technique is to improve the accuracy $\overline{\varepsilon}$ of the approximation until the correctly rounded value can be decided. Given a function f and an argument x, the value of $f(x)$ is first evaluated using a quick approximation of accuracy $\overline{\varepsilon}_1$. Knowing $\overline{\varepsilon}_1$, it is possible to decide if rounding is possible, or if more accuracy is required, in which case the computation is restarted using a slower approximation of accuracy $\overline{\varepsilon}_2$ greater than $\overline{\varepsilon}_1$, and so on. This approach leads to good average performance, as the slower steps are rarely taken.

1.3 Improving on Ziv's approach

However there was until recently no practical bound on the termination time of Ziv's iteration: It may be proven to terminate for most transcendental functions, but the actual maximal accuracy required in the worst case is unknown. In `libultim`, the measured worst-case execution time is indeed three orders of magnitude larger than that of usual `libm`s (see Table 2 below). This might prevent using this library in critical applications. A related problem is memory requirement, which is, for the same reason, unbounded in theory, and much higher than usual `libm`s in practice.

Probably for this reason, Ziv's implementation doesn't provide a proof of the correct rounding property, and indeed several functions fail to return the correctly rounded result for some input values (although most of these errors have been corrected in the version which is part of the GNU `glibc`). Sun's library doesn't provide a proof, either.

Finally, IBM's library lacks the directed rounding modes (Sun's library does provide them). These rounding modes might be the most useful: Indeed, in round-to-nearest mode, correct rounding provides an accuracy improvement over usual `libm`s of only a fraction of a unit in the last place *(ulp)*, since the values difficult to round were close to the middle of two consecutive floating-point numbers. This may be felt of little practical significance. However, the three other rounding modes are needed to guarantee intervals in interval arithmetic. Without correct rounding in these directed rounding modes, interval arithmetic may loose up to two *ulp* of precision in each computation. Actually, current interval elementary function libraries are even less accurate than that, because they sacrifice accuracy to a very strict proof [18].

The goal of the `crlibm` (Correctly Rounded `libm`) project is therefore a library which is

- correctly rounded in the four IEEE-754 rounding modes,
- proven,
- and sufficiently efficient in terms of performance (both average and worst-case) and resources (in particular we impose an upper bound of 4KB of memory consumed per function [6])

to enable the standardization of correct rounding for elementary functions.

1.4 Organisation of this article

Section 2 describes the general principles of the `crlibm` library, from the theoretical aspects to an implementation framework which makes optimal use of current processor technology, and a proof framework which is currently a distinctive feature of this work. Section 3 is an in-depth

example of using these frameworks to implement an efficient, proven correctly-rounded natural logarithm. Section 4 gives measures of performance and memory consumption and shows that this implementation compares favorably to the best available accurate-faithful `libm`s on most architectures.

2 The Correctly Rounded Mathematical Library

2.1 Worst cases for correct rounding

Lefèvre and Muller [23, 21] computed the worst-case $\overline{\varepsilon}$ required for correctly rounding several functions in double-precision over selected intervals in the four IEEE-754 rounding modes. For example, they proved that 157 bits are enough to ensure correct rounding of the exponential function on all of its domain for the four IEEE-754 rounding modes, and 118 bits for the logarithm. Up-to-date information about this quest for worst cases (which functions are covered on which interval) is available in the documentation of `crlibm`[1]. A discussion of the possible strategies in the absence of worst cases is also available in this document.

2.2 Two steps are enough

Thanks to such results, we are able to guarantee correct rounding in two steps only, which we may then optimize separately. The first *quick* step is as fast as current `libm`, and provides an accuracy between 2^{60} and 2^{80} (depending on the function), which is sufficient to round correctly to the 53 bits of double precision in most cases. The second *accurate* step is dedicated to challenging cases. It is slower but has a reasonably bounded execution time, being tightly targeted at Lefèvre/Muller worst cases (contrary to Sun's and IBM's library).

2.3 On portability and performance

`crlibm` was initially a strictly portable library, relying only on two widespread standards: IEEE-754 for floating-point, and C99 for the C language. This meant preventing the compiler/processor combination from using advanced floating-point features available in recent mainstream processors, and as a consequence accepting a much lower performance than the default, accurate-faithful `libm`, typically by a factor 2 [13, 10].

Among these advanced features, the most relevant to the implementation of elementary functions are:

- hardware double-extended (DE) precision, which provides 64 bits of mantissa instead of the 53 bits of the IEEE-754 double format,
- hardware fused multiply-and-add (FMA), which performs the operation $x \times y + z$ in one instruction, with only one rounding.

It was suggested that a factor two in performance would be an obstacle to the generalization of correct rounding, therefore our recent research has focussed on exploiting these features. The logarithm is the first function which has been completed using this approach: In versions of `crlibm` strictly greater than 0.8, there is a compile-time selection between two implementations.

- The first exploits double-extended precision if available (for ia32 and ia64 processors), and is referred to as the "DE" version in the following.
- The second relies on double-precision only, and is referred to as the "portable" version in the following.

Both versions exploit an FMA if available (on Power/PowerPC essentially for the portable version, on Itanium for the DE version). In the absence of an FMA, the portable version is strictly portable in the IEEE-754/C99 sense. This choice provides optimized versions (as section 4 will show) for the overwhelming majority of current mainstream processors.

2.4 Fast first step

The DE version of the first step is very simple: as double-extended numbers have a 64-bit mantissa, it is easy to design algorithms that compute a function to an accuracy better than 2^{-60} using only DE arithmetic [25].

For the portable version, we only have double-precision at our disposal. We classically represent a number requiring higher precision (such as y_1, the result of the first step) as the sum of two floating-point numbers, also called a *double-double* number. There are well-known algorithms for computing on double-doubles [14].

In both versions, we also make heavy use of classical, well proven results like Sterbenz' lemma [17] which gives conditions for a floating-point subtraction to entail no rounding error.

2.5 Rounding test

At the end of the fast step, a sequence of simple tests on y_1 either returns a correctly rounded value, or launches the second step. We call such a sequence a *rounding test*. The property that a rounding test must ensure is the following: a value will be returned only if it can be proven to be the correctly rounded value of $\hat{y}$, otherwise (in doubt) the second step will be launched.

A rounding test depends on a bound $\overline{\varepsilon}_1$ on ε_1, the overall relative error of the first step. This bound is usually computed statically, although in some case it can be refined at runtime (IBM's code has such *dynamic* rounding tests, but for an explained and proven example see `crlibm`'s tangent [1]). Techniques for computing $\overline{\varepsilon}_1$, as well as techniques for proving the validity of a rounding test, will be detailed in Section 2.9.

The implementation of a rounding tests depends on the rounding mode and the nature of y_1 (a double-extended for the DE version, or a double-double for the portable version). Besides, in each case, there are several sequences which are acceptable as rounding tests. Some use only floating point but require pre-computing on $\overline{\varepsilon}_1$ [10], others first extract the mantissa of y_1 and perform bit mask operations on the bits after the 53rd [1]. All these possible tests are cleanly encapsulated in C macros.

2.6 Accurate second step

For the second step, correct rounding needs an accuracy of 2^{-120} to 2^{-150}, depending on the function. We are now using three different approaches depending on the processor's capabilities.

- We have designed an ad-hoc multiple-precision library called `scslib` which is lightweight, very easy to use in the context of `crlibm`, and more efficient than all other available comparable libraries [12, 7]. It allows quick development of the second step, and has been used for the initial implementation of all the functions. It is based on integer arithmetic.

- For the DE version of the second step, it has been proven that *double-double-extended* intermediate computations are always enough to ensure correct rounding, even when worst cases have been found requiring more than the 128 bits of precision offered by this representation [8]. Using double-double-extended is not as simple as using `scslib`, however the algorithms are those already used and proven for double-double. And it is much more efficient than `scslib`: we measure a factor 10 in the worst-case execution time [9].

- Finally, we are developping portable second steps based on *triple-double* arithmetic. This approach is also much more efficient than `scslib`, but it is also much more difficult to use and to prove. The logarithm presented below is the first function to be implemented using this technology.

The main reason for the performance improvement over `scslib` is that each computation step can use the required precision, no more. Typically for instance we start a Horner polynomial evaluation in double, continue in double-double, and perform only the last few iterations in triple double. The `scslib` format doesn't offer this flexibility. Another advantage is that the accurate

step can use table-based methods [15, 16, 29] because triple-double is much less memory-consuming than the `scslib` format, all the more as these tables can be shared with the first step, as will be seen in Section 3.

The main advantage of using `scslib` is that it leads to very easy error computations. However, being based on integer arithmetic, `scslib` is also interesting for architectures without floating-point hardware.

2.7 Final rounding

The result of the accurate step will be either a triple-double number, or a double-double-extended number, or a number represented in `scslib`'s special format. This result must finally be rounded to a double-precision number in the selected rounding mode. This operation is peculiar to each of the three representations mentioned.

- The functions provided by `scslib` for this purpose are very straightforward, but quite slow.
- Processors which support double-extended precision are all able to round the sum of two double-extended numbers to a double, in an atomic operation. Fortunately, this is even easy to express in C [19] as
  `return (double)(yh+yl);` where `yh` and `yl` are double-extended numbers. Note however that more care must be taken for functions whose worst cases may require more than 128 bits [8].
- The most difficult case is that of the triple-double representation, because it is a redundant representation, and because there is no hardware for doing a ternary addition with only a final rounding [11]. Again, we have designed sequences of operations for this final rounding in the four rounding modes. These sequences involve a dozen of floating-point operations and nested tests, and their full proof is several pages long [1].

2.8 Error analysis and the accuracy/performance tradeoff

The probability p_2 of launching the second (slower) step is the probability that the interval $[y(1-\overline{\varepsilon}), y(1+\overline{\varepsilon})]$ contains the middle of two consecutive floating-point numbers (or a floating-point number in directed rounding modes). Therefore, it is obviously proportional to the error bound $\overline{\varepsilon}_1$ computed for the first step.

This defines the main performance tradeoff one has to manage when designing a correctly-rounded function: The average evaluation time will be

$$T_{\text{avg}} = T_1 + p_2 T_2 \tag{1}$$

where T_1 and T_2 are the execution time of the first and second phase respectively (with $T_2 \approx 100T_1$ in `crlibm` using `scslib`, and $T_2 \approx 10T_1$ in `crlibm` using DE or triple-double), and p_2 is the probability of launching the second phase.

Typically we aim at chosing (T_1, p_2, T_2) such that the average cost of the second step is negligible: This will mean that the cost of correct rounding is negligible. The second step is built to minimize T_2, there is no tradeoff there. Then, as p_2 is almost proportional to $\overline{\varepsilon}_1$, to minimize the average time, we have to

- balance T_1 and p_2: this is a performance/precision tradeoff (the more accurate the first step, the slower)
- and compute a tight bound on the overall error $\overline{\varepsilon}_1$.

Computing this tight bound is the most time-consuming part in the design of a correctly-rounded elementary function. The proof of the correct rounding property only needs a proven bound, but a loose bound will mean a larger p_2 than strictly required, which directly impacts

average performance. Compare $p_2 = 1/1000$ and $p_2 = 1/500$ for $T_2 = 100T_1$, for instance. As a consequence, when there are multiple computation paths in the algorithm, it may make sense to precompute different values of $\overline{\varepsilon}_1$ on these different paths [10].

2.9 Proving correct rounding

With this two-step approach, proving the correct-rounding property resumes to two tasks:

- Computing a bound on the overall error of the second step, and checking that this bound is less than Lefèvre/Muller worst-case accuracy;
- Proving that the first step returns a value only if this value is correctly rounded, which also requires a proven (and tight) bound on the evaluation of the first step.

One difficulty is that the smallest change in the code (for optimization, and even for a bug fix) will affect the proof. We therefore strive to factorize our code in a way compatible with proof-writing. For example, elementary function are typically implemented using polynomial approximation techniques. The latter can finally be based on addition and multiplication for the different formats. For proof purposes, we want to consider e.g. an addition of two triple-double numbers just like a floating point addition with its respective "machine epsilon". The challenge here is the tradeoff between efficiency and provability.

Therefore we structure our code and proof as follows. Our basic procedures (including addition and multiplication for various combinations of double, double-double and triple-double arguments and results, but also rounding tests, final rounding, etc.) are implemented as C macros. These macros may have different processor-specific implementation, for example to use an FMA when available, so this solution provides both flexibility and efficiency. A non-exhaustive list of some properties of these procedures is given by table 1.

Name	Operation	Property
`Add12`	$x_h + x_l = a + b$	exact (Fast2Sum)
`Mul12`	$x_h + x_l = a \cdot b$	exact (Dekker)
`Add22`	$x_h + x_l \approx (a_h + a_l) + (b_h + b_l) + \delta$	$\varepsilon \leq 2^{-103.5}$
`Mul22`	$x_h + x_l \approx (a_h + a_l) \cdot (b_h + b_l) + \delta$	$\varepsilon \leq 2^{-102}$
`Add33`	$x_h + x_m + x_l \approx$ $(a_h + a_m + a_l) + (b_h + b_m + b_l) + \delta$	ε depends on overlap
`Add233`	$x_h + x_m + x_l \approx$ $(a_h + a_l) + (b_h + b_m + b_l) + \delta$	ε depends on overlap
`Mul23`	$x_h + x_l \approx (a_h + a_l) \cdot (b_h + b_l) + \delta$	$\varepsilon \leq 2^{-149}$
`Mul233`	$x_h + x_l \approx$ $(a_h + a_l) \cdot (b_h + b_m + b_l) + \delta$	ε depends on overlap
`Renormalize3`	$x_h + x_m + x_l = a_h + a_m + a_l$	no overlap in result
`RoundToNearest3`	$x' = \circ(x_h + x_m + x_l)$	(Round-to-nearest)
`RoundUp3`	$x' = \triangle(x_h + x_m + x_l)$	(Round-upwards)
`TEST_AND_RETURN_RN`	`return` $\circ(x_h + x_l)$ if rounding is safe knowing ε `continue` otherwise	

Table 1: Some basic procedures encapsulated in C-Macros

Each macro has a proof published in [1] and covering all its implementations. What we prove is actually a theorem with hypotheses (validity conditions for the macro to work), and a conclusion (the relative error of the operator is smaller than some epsilon, for example). The proof of a function invoking such a macro will then have to check the hypotheses, and may then use the conclusion.

2.10 Using automated error analysis tools

We strive to automate the error computation to make proofs easier and increase confidence in the result. A first approach is to rely on clean Maple scripts to compute the numerical constants, output the C header files containing these constants, and implement the error computation out of them. Of course, these scripts are part of the `crlibm` distribution.

More recently, we have been making increasing use of Gappa, a tool which manages ranges and errors in numerical code using interval arithmetic [5]. This tool takes a code fragment, information on the inputs (typically their ranges and bounds on their approximation errors), and computes and propagates roundoff errors. It is far from being automatic: The user has to provide almost all the knowledge that would go in a paper proof, but does so in an interactive and very safe way, increasing the confidence that all the contribution to the total error are taken properly into account. Besides, this tool relies on a library of theorems which take into account subnormal numbers, exceptional cases, etc, which ensures that theses exceptional cases are considered. This tool outputs a proof in the Coq language [3], and this proof can be machine-checked provided all the support theorems have been proven in Coq. Obviously, our divide-and-conquer approach matches this framework nicely, although we currently don't have Coq proofs of all the previous theorems.

Ultimately, we hope that the "paper" part of the proof will be reduced to an explanation of the algorithms and of the structure of the proof. One of the current weakest point is the evaluation of infinite norms (between an approximation polynomial and the function), which we do in Maple. As we approximate elementary functions on domains where they are regular and well-behaving, we can probably trust Maple here, but a current research project aims at designing a validated infinite norm. Another approach is to rely on Taylor approximations with carefully rounded coefficients, such that mathematical bounds on the approximation error can be computed [18]. The main drawback is that it typically leads to polynomials of higher degree for the same approximation error, which results in larger delays, larger memory consumption, and possibly larger rounding errors.

3 `crlibm`'s correctly rounded logarithm function

This section is a detailed example of the previous approach and framework.

3.1 Overview

The worst-case accuracy required to compute the natural logarithm correctly rounded in double precision is 118 bits according to Lefèvre and Muller [22]. The first step is accurate to 2^{-60}, and the second step to 2^{-120}, for all the implementations.

For the quick phase we now use a different algorithm as the one presented in [10]. This choice is motivated by two main reasons:

- The algorithm is slightly more complex, but faster.
- It can be used for all our different implementations (portable or DE).

Special cases are handled in all implementations as follows: The natural logarithm is defined over positive floating point numbers. If $x \leq 0$, then $\log(x)$ should return NaN. If $x = +\infty$, then $\log(x)$ should return $+\infty$. This is true in all rounding modes.

Concerning subnormals, the smallest exponent for an non-zero logarithm of a double-precision input number is -53 (for the input values $\log(1+2^{-52})$ and $\log(1-2^{-52})$, as $\log(1+\varepsilon) \approx \varepsilon$ when $\varepsilon \to 0$). As the result is never subnormal, we may safely ignore the accuracy problems entailed by subnormal numbers.

The common algorithm is inspired by the hardware based algorithm proposed by Wong and Goto[30] and discussed further in [28]. After handling of special cases, consider the argument x written as $x = 2^{E'} \cdot m$, where E' is the exponent of x and m its mantissa, $1 \leq m < 2$. This

decomposition of x into E' and m can be done simply by some integer operations. In consequence, one gets $\log(x) = E' \cdot \log(2) + \log(m)$. Using this decomposition directly would lead to catastrophic cancellation in the case where $E' = -1$ and $m \approx 2$. Therefore, if m is greater than approximately $\sqrt{2}$, we adjust m and E as follows:

$$E = \begin{cases} E' & \text{if} \quad m \leq \sqrt{2} \\ E'+1 & \text{if} \quad m > \sqrt{2} \end{cases} \qquad y = \begin{cases} m & \text{if} \quad m \leq \sqrt{2} \\ \frac{m}{2} & \text{if} \quad m > \sqrt{2} \end{cases}$$

All of the operations needed for this adjustment can be performed exactly. We see that y is now bounded by $\frac{\sqrt{2}}{2} \leq y \leq \sqrt{2}$ leading to a symmetric bound for $\log(y)$.

The magnitude of y being still too big for polynomial approximation, a further argument reduction is done as follows. The algorithm looks up, using the high magnitude bits of the mantissa of y, a value r_i which approximates relatively well $\frac{1}{y}$. Setting $z = y \cdot r_i - 1$, we obtain for the reconstruction

$$\log(x) = E \cdot \log(2) + \log(1+z) - \log(r_i)$$

Now z is small enough (typically $|z| < 2^{-7}$) for approximating $\log(1+z)$ by a Remez polynomial $p(z)$. The values for $\log(2)$ and $\log(r_i)$ are tabulated.

One crucial point here is the operation sequence for calculating z out of y and r_i: $z = y \cdot r_i - 1$. In the DE code, the r_i are chosen as floating-point numbers with at most 10 non-zero consecutive bits in their mantissa (they are actually tabulated as single-precision numbers). As y is a double-precision number, the product $y \cdot r_i$ fits in a double-extended number, and is therefore computed exactly in double-extended arithmetic. The subtraction of 1 is then also exact thanks to Sterbenz' lemma: $y \cdot r_i$ is very close to 1 by construction of r_i. Finally the whole range reduction involves no roundoff error, which will of course ease the proof of the algorithm.

In the portable version, there is unfortunately no choice of r_i that will guarantee that $y \cdot r_i - 1$ fits in one double-precision number. Therefore we perform this computation in double-double, which is slower, but still with the guarantee that z as a double-double is exactly $z = y \cdot r_i - 1$.

This algorithm allows for sharing the tables between the first and the second step: In the portable version, these tables are normalized triple-double, the first step using only two of the three values. In the DE version, the tables are double-double-extended, and the first step uses only one of the two values. Such sharing brings a huge performance improvement over the previous approach [10] where the two steps were using two distinct algorithms and different tables. This improvement has two causes. First, the second step does not need to restart the computation from the beginning. As the argument reduction is exact, the second step doesn't need to refine it. Second, the table cost of the second step is much reduced, which allows more entries to the table, leading to a smaller polynomial (especially for the second step). Here, tables are composed of 128 entries $(r_i, \log(r_i))$.

3.2 A double-extended logarithm

The double-extended version is a straightforward implementation of the previous ideas: Argument reduction and first step use only double-extended arithmetic, second step uses a polynomial of degree 14, evaluated in Horner form, with the 8 lower degrees implemented in double-double-extended arithmetic.

The most novel feature of this implementation is the proof of a tight error bound for the Estrin parallel evaluation [28] of the polynomial of the first step, given below.

```
z2=z*z;  p67=c6+z*c7;  p45=c4+z*c5;  p23=c2+z*c3;  p01=logirh+z;
z4=z2*z2;   p47=p45+z2*p67;   p03=p01+z2*p23;
p07=p03+z4*p47;
log=p07+E*log2h;
```

Here we have written on one line the operations that can be computed in parallel: Such an evaluation scheme is increasingly relevant to superscalar, deeply pipelined processors (see [4] for some quantitative aspects).

However it is much more difficult to compute a tight error bound on such code than on a more classical Horner evaluation. We show that the overall relative error is smaller than 2^{-61} in all cases. As soon as $|E|$ is larger than 2, this bound is relatively easy to prove by computing a bound on the absolute error, and dividing by the minimum value of the final logarithm. However, for the cases where the log comes close to zero, the proof is much more complex: The error computation has to combine relative errors only, and has to use the knowledge that $\log(1+x) \approx x$. It would probably not have been possible without machine assistance.

The full proof is available in the `crlibm` distribution.

3.3 A portable logarithm

The quick phase uses a modified minimax polynomial of degree 7. The polynomial $q(z_h)$ consisting of its monomials of degrees 3 to 7 is evaluated using Horner's scheme. The full polynomial, to be evaluated as a double-double $p_h + p_l$, is given by the following expression:

$$p(z) \approx z - \frac{1}{2} \cdot z^2 + z^3 \cdot q(z) \tag{2}$$

where z stands for the double-double $z_h + z_l$. The different summands are computed as follows:

1. We approximate z^2 by
$$z^2 = (z_h + z_l)^2 \approx z_h^2 + 2 \cdot z_h \otimes z_l$$
(here $\otimes$ denotes the machine operation)

2. z_h^2 is computed exactly using a `Mul12` macro, and multiplied exactly by 1/2. An `Add22` then adds this result to the first-order term $z_h + z_l$.

3. the term $1/2 \cdot 2 \cdot z_h \otimes z_l$ is computed as a single double precision multiplication, and will actually be added to $z^3 \cdot q(z)$ at step (5).

4. For the less significant terms, $q(z)$ is approximated as $q(z_h)$, and z^3 is also approximated:
$$z^3 \approx (z_h \otimes z_h) \otimes z_h$$
where $z_h \otimes z_h$ has already been computed in the previous `Mul12`.

5. The result for $z^3 \cdot q(z_h)$ will be in double precision. An `Add12` builds a double-double approximating $z^3 \cdot q(z_h) - z_h \cdot z_l$

6. Finally, the double-doubles obtained at steps (2) and (5) are added together using an `Add22`.

The double-double $p_h + p_l$ thus obtained is then added to the tabulated values $lr_h + lr_m \approx \log(r_i)$ and $l2_h + l2_m \approx \log(2)$, the latter being multiplied by E. Again in this multiplication we use the fact that E is a small integer requiring no more than 12 bits to implement an accurate multiplication without resorting to heavier higher precision arithmetic: $l2_h$ is tabulated with a mantissa of 40 bits only.

The final error of of about 2^{-60} bits is mainly due to the relative approximation error which is about 2^{-62} and the roundoff errors, about 2^{-61}. The latter are reigned by the omission of z_l in the higher degree monomials. Further analysis on these issues and the proof of the function can be found in [1].

The polynomial for accurate phase is of degree 14. It is evaluated for degree 3 through 14 using Horner's scheme. Double-precision arithmetic is used for degree 10 through 14 omitting z_l. Double-double arithmetic – taking into account also z_l – takes over for degrees 3 to 9. The lower degree monomials are evaluated in an ad hoc scheme. We can so limit the use of costly

triple-double arithmetic to the absolute minimum. For example we never have to multiply two triple-doubles.

The reconstruction phase uses here all three of the values in the tables. It consists in 3 full triple-double additions. Special care is needed in the proof for the final addition of the value representing $E \cdot \log(2)$, because we compute $E \cdot \log(2)$ as an overlapping triple-double (again because $l2_h$ and $l2_l$ have 40-bit mantissa to allow for a fast computation of $E \cdot \log(2)$). The final rounding uses the macro already presented.

Notwithstanding its portability, the triple-double based implementation can be accelerated by the optional use of FMA-instructions; the gain in performance will be analyzed in section 4. This decision is made at compile time.

4 Analysis of the logarithm performance

The input numbers for the performance tests given here are random positive double-precision numbers with a normal distribution on the exponents. More precisely, we take random 63-bit integers and cast them into double-precision numbers.

On average, the second step is taken less than 1% of the calls in all the implementations, which ensures negligible average overhead of the second step, considering the respective costs of the first and second steps (see the tables below).

4.1 Speed

Table 2 compares the absolute timings of logarithm implementations from a variety of libraries on a Pentium 4.

Library	avg time	max time
Sun's `libmcr`	1055	831476
IBM's `libultim`	677	463488
`crlibm` portable using `scslib`	706	55804
`crlibm` portable using triple-double	634	5140
`crlibm` using double-extended	339	4824
default `libm` (without correct rounding)	*323*	*8424*

Table 2: Comparisons of double-precision logarithm implementations from a range of libraries (times in cycles on a Pentium 4 Xeon / Linux Debian 3.1 / gcc 3.3)

Table 3 compares timings for a variety of processors, all on a recent Linux with gcc3.3 compiler. This table clearly shows that our correctly-rounded logarithm has average performance comparable to the default `libm`. Note that the default library on the Power/PowerPC architecture on Linux is derived from IBM's `libultim`, and offers correct rounding, although unproven.

These tables also show the benefit of having a second step specifically written to match the worst case required accuracy: we have a worst case execution time almost two orders of magnitude better than the other correctly rounded libraries `libmcr` and `libultim`.

4.2 Memory requirements

In both the double-extended and the portable versions, we have a 128-entry table, each entry consisting of a float (on 4 bytes) and either a triple-double (24 bytes) or a double-double-extended (20 bytes). Table sizes are therefore

- $128 \times (4 + 3 \times 8) = 3584$ bytes for the portable version, and

Opteron (cycles)	avg time	max time
`crlibm` using double-extended	118	862
default `libm` (without correct rounding)	*189*	*8050*
Pentium 4 (cycles)	avg time	max time
`crlibm` using double-extended	339	4824
default `libm` (without correct rounding)	*323*	*8424*
Pentium 3 (cycles)	avg time	max time
`crlibm` using double-extended	150	891
default `libm` (without correct rounding)	*172*	*1286*
Power5 (arbitrary units)	avg time	max time
`crlibm` (without FMA)	50	259
`crlibm` (using FMA)	42	204
default libm (IBM's `libultim`)	52	28881
Itanium 1 (cycles)	avg time	max time
`crlibm` using double-extended and FMA	73	2150
default `libm` (without correct rounding)	*54*	*8077*

Table 3: `crlibm` versus default `libm` on a range of processors

- $128 \times (4 + 2 \times 10) = 3072$ bytes for the double-extended version.

Polynomial coefficient values and rounding constants are directly compiled into the code. Note that the actual memory usage may be larger on 64-bit processors, due to memory alignment constraints. The most wasteful is the Itanium, where each floating-point number must be aligned on a 64-bit (or 8 bytes) boundary: Therefore a single-precision number consumes 8 bytes, and a double-extended consumes 16 bytes, leading to a table of actual size $128 \times (8 + 2 \times 16) = 5120$ bytes. This is not so much of a a problem, thanks to the huge caches of Itanium processors.

The previous version of the portable logarithm, using `scslib`, used 1KB of tables for the first step and 2KB for the second step. Note however that the `scslib` format was quite wasteful in terms of code size. For instance, on a Pentium, the total code size was about 5KB whereas it is about 2KB for the DE version. In both cases, the first step is compiled in about 500 bytes of code. These numbers are provided by the Unix `objdump` command, and similar numbers are obtained for other architectures.

On this respect we should mention to be fair that the `log` from the default `libm` on x86 architectures consumes very little memory (40 bytes altogether!) since it uses the machine instructions specifically designed for the evaluation of logarithms: `fyl2xp1` and `fyl2x`. These instructions consume silicon in the processor, but no additional memory. With current technology, as already mentionned, it makes more sense to use this silicon to accelerate more general computations, and delegate elementary functions to software. This explains why modern instruction sets do not offer the equivalent of `fyl2xp1` and `fyl2x`. As an illustration, on two x86 processors out of three, our implementation of the logarithm is faster than the one using these machine instructions, probably because it may use large amounts (4KB) of tables in inexpensive memory.

5 Conclusion and perspectives

This article presents an implementation of the natural logarithm in double precision which has a unique combination of ambitious features:

- It offers correct rounding, the best accuracy that is mathematically possible considering the finite nature of double-precision.
- It is portably optimized, exploiting processor-specific features through a high-level language, relying only to compliance to the C99 standard. Its performance matches that of the best available standard mathematical libraries.
- It intends to be proven, with the help of state-of-the-art tools for machine-assisted automatic theorem proving.

In these respects, this function is the most advanced among the current distribution of `crlibm` (version 0.8), which counts 8 functions developed over two years. Short-term efforts include writing new functions, but also improving the quality of the other functions, in particular improving and machine-checking the proofs and writing triple-double second steps to replace the current code based on `scslib`. A longer-term research goal is to keep increasing confidence in the proofs. The `crlibm` framework is also well suited to the implementation of "perfect" interval functions (fficient, as tight as mathematically possible, and proven).

The complete code with a more detailed proof, including the Maple programs and the Gappa proofs mentioned above, is available from [1], with a warning: some of it may be only in the CVS repository which is also publicly accessible.

References

[1] CR-Libm, a library of correctly rounded elementary functions in double-precision. http://lipforge.ens-lyon.fr/www/crlibm/.

[2] Standard 754-1985 for binary floating-point arithmetic (also IEC 60559), 1985. ANSI/IEEE.

[3] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development. Coq'Art: the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[4] M. Cornea, J. Harrison, and P.T.P Tang. *Scientific Computing on Itanium-based Systems*. Intel Press, 2002.

[5] Marc Daumas and Guillaume Melquiond. Generating formally certified bounds on values and round-off errors. In *6th Conference on Real Numbers and Computers*, pages 55–70, Schloss Dagstuhl, Germany, November 2004.

[6] D.Defour. Cache-optimised methods for the evaluation of elementary functions. Technical Report RR2002-38, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 2002.

[7] F. de Dinechin and D. Defour. Software carry-save: A case study for instruction-level parallelism. In *Seventh International Conference on Parallel Computing Technologies*, Nizhny Novgorod, Russia, September 2003.

[8] F. de Dinechin, D. Defour, and C. Lauter. Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Technical Report 2004-10, LIP, École Normale Supérieure de Lyon, March 2004.

[9] F. de Dinechin, A. Ershov, and N. Gast. Towards the post-ultimate libm. In *17th IEEE Symposium on Computer Arithmetic*, Cape Cod, Massachussets, June 2005.

[10] F. de Dinechin, C. Loirat, and J.-M. Muller. A proven correctly rounded logarithm in double-precision. In *RNC6, Real Numbers and Computers*, Schloss Dagstuhl, Germany, November 2004.

[11] D. Defour. Collapsing dependent floating point operations. Technical report, DALI Research Team, LP2A, University of Perpignan, France, December 2004.

[12] D. Defour and F. de Dinechin. Software carry-save for fast multiple-precision algorithms. In *35th International Congress of Mathematical Software*, Beijing, China, 2002.

[13] D. Defour, F. de Dinechin, and J.-M. Muller. Correctly rounded exponential function in double precision arithmetic. In *Advanced Signal Processing Algorithms, Architectures, and Implementations X (SPIE'2000)*, San Diego, California, August 2001.

[14] T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[15] P. M. Farmwald. High bandwidth evaluation of elementary functions. In *Proceedings of the 5th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, 1981.

[16] S. Gal. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations, LNCS 235*, pages 1–16. Springer Verlag, 1986.

[17] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991.

[18] W. Hofschuster and W. Krämer. FI_LIB, eine schnelle und portable Funktionsbibliothek für reelle Argumente und reelle Intervalle im IEEE-double-Format. Technical Report Nr. 98/7, Institut für Wissenschaftliches Rechnen und Mathematische Modellbildung, Universität Karlsruhe, 1998.

[19] ISO/IEC. *International Standard ISO/IEC 9899:1999(E). Programming languages – C*. 1999.

[20] R. Klatte, U. Kulisch, C. Lawo, M. Rauch, and A. Wiethoff. *C-XSC a C++ class library for extended scientific computing*. Springer Verlag, 1993.

[21] V. Lefèvre. *Moyens arithmétiques pour un calcul fiable*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2000.

[22] V. Lefèvre and J.-M Muller. Worst cases for correct rounding of the elementary functions in double precision. http://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm, 2004.

[23] V. Lefèvre, J.M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, November 1998.

[24] IBM Accurate Portable MathLib. http://oss.software.ibm.com/mathlib/.

[25] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN: 0130183482.

[26] R.E. Moore. *Interval analysis*. Prentice Hall, 1966.

[27] MPFR. http://www.mpfr.org/.

[28] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.

[29] P. T. P. Tang. Table lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, June 1991. IEEE Computer Society Press.

[30] W. F. Wong and E. Goto. Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *IEEE Transactions on Computers*, 43(3):278–294, March 1994.

[31] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991.