Laboratoire de l'Informatique du Parallélisme

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

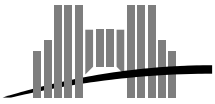
Towards the post-ultimate `libm`

Florent de Dinechin,
Nicolas Gast

November 2004

Research Report N° 2004-47

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`



Towards the post-ultimate `libm`

Florent de Dinechin, Nicolas Gast

November 2004

Abstract

This article presents advances in the subject of double-precision correctly rounded elementary functions since the publication of the `libultim` mathematical library developed by Ziv at IBM. This library demonstrated that the performance overhead of correct rounding could be made negligible in average. However, the worst case execution time was up to 1000 times the average time, and memory consumption was also a problem. To address these questions, a range of new techniques, from the more portable to the more efficient, are presented, and demonstrated on two typical functions, exponential and arctangent. The main result of this paper is to show that the worst-case execution time can be bounded within a factor of 2 to 10 of the average time, with memory consumption comparable to current `libm`s. This has in turn implications on the techniques and tradeoffs for correctly rounded functions. This article also shows that these techniques make it much easier to prove the correct rounding property. Thus, this article lifts the last technical obstacles to a widespread use of (at least some) correctly rounded double precision elementary functions.

Keywords: elementary functions, correct rounding, IEEE-754

Résumé

Cet article présente les progrès réalisés dans l'implémentation des fonctions élémentaires avec arrondi correct en double précision depuis la bibliothèque `libultim` réalisée par Ziv chez IBM. Cette bibliothèque avait démontré que le surcoût moyen de l'arrondi correct pouvait être rendu négligeable, toutefois le temps d'exécution au pire cas pouvait être plusieurs milliers de fois le temps moyen, avec une consommation mémoire à l'avenant. Pour résoudre ce problème, cet article présente une palette de techniques, certaines portables et certaines utilisant au mieux les spécificités des processeurs récents. Ces techniques sont testées pour deux fonctions, l'exponentielle et l'arctangente. Le résultat principal est que le temps au pire cas peut être réduit à un facteur 2-10 du temps moyen, avec une consommation mémoire similaire à celle des `libm` courantes. Ceci a des implications sur les techniques et les compromis d'implémentation utilisées pour garantir l'arrondi correct. Cela rend également plus facile la preuve de la propriété d'arrondi correct. Pour toutes ces raisons, cet article lève les derniers obstacles à une utilisation généralisée d'au moins quelques fonctions élémentaires avec arrondi correct.

Mots-clés: fonctions élémentaires, arrondi correct, IEEE-754

1 Introduction

1.1 Correct rounding and elementary functions

The need for accurate elementary functions is important in many critical programs. Methods for computing these functions include table-based methods[10, 24], polynomial approximations and mixed methods[4]. See the books by Muller[22] or Markstein[18] for recent surveys on the subject.

The IEEE-754 standard for floating-point arithmetic[3] defines the usual floating-point formats (single and double precision) and specifies precisely 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: the result should be as if the calculation had been performed in infinite precision, and then rounded.

The adoption and widespread use of the IEEE-754 standard 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. Directed rounding modes (towards $+\infty$, $-\infty$ and 0) are also the key to enable efficient *interval arithmetic* [20, 13].

However, the IEEE-754 standard specifies nothing about elementary functions, which limits these advances to code excluding such functions. Without a standard, until recently, only two options were available:

- Using the mathematical libraries (`libm`) provided by operating systems, which are efficient but do not garantee correct rounding. Older `libm`s give no guarantee at all, but the most recent ones return a result with an error smaller than one unit in the last place (ulp) and with a high probability of correct rounding. We will compare our performance to such libraries, which we call *accurate-faithful* in the following;

- Using multiple-precision packages like `mpfr` [21] which offer correct rounding in all rounding modes, but are several orders of magnitude slower than the `libm` for the same precision (see the tables in Sec. 4).

This situation changed with the publication of the IBM Accurate Portable Mathlib [17] (or `libultim`), which offers most usual functions correctly rounded to the nearest, and average performance similar to existing `libm`s. The method used in this library, published by Ziv [25], is the following.

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

In most cases, the image $\hat{y}$ of a floating-point number x by an elementary function f is not a floating point number, and can therefore not be represented exactly in standard numeration systems. The purpose here is to compute the floating-point number that is closest to (resp. immediately above or immediately below) this mathematical value, which we call the result *correctly rounded* to the nearest (resp. towards $+\infty$ or towards $-\infty$).

A computer may evaluate an approximation y to the real number $\hat{y}$ with precision $\overline{\varepsilon}$. 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) [22].

Ziv's technique is to improve the precision $\overline{\varepsilon}$ of the approximation until the correctly rounded value can be decided. Given a function f and an argument x, a first, quick approximation y_1 to the value of $f(x)$ is evaluated, with accuracy $\overline{\varepsilon}_1$. Knowing $\overline{\varepsilon}_1$, it is possible to decide if it is possible to round y_1 correctly, or if more precision is required, in which case the computation is restarted using a slower approximation of precision $\overline{\varepsilon}_2$ better 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 precision required in the worst case is unknown. According to a classical statistical arguments by Gal [11, 22], and assuming the implementation can be proven correct, which is a huge problem in itself, the `libultim` approach provides correct rounding with probability higher than $1 - 2^{-500}$, which we call *astronomical confidence* in the following. However the need for arbitrary multiple precision has also a cost in terms of performances: In `libultim`, the measured worst-case execution time is indeed three orders of magnitude higher than that of accurate-faithful `libm`s. This might prevent using this method in critical application. 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 these reasons, Ziv's implementation doesn't provide a proof of the correct rounding property.

Finally, this library still lacks the directed rounding modes, which might be the most useful: Indeed, correct rounding provides a precision improvement over usual `libm`s of only a fraction of a unit in the last place *(ulp)* in round-to-nearest mode. 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 looses up to one *ulp* of precision in each computation.

The goal of the `crlibm` project (at `http://lipforge.ens-lyon.fr/projects/crlibm/`) is therefore to design a mathematical library which is

- portable to any system implementing the ISO-C99 and IEEE-754 standards,
- correctly rounded in the four IEEE-754 rounding modes,
- proven, both theoretically and in the implementation,
- and reasonably efficient in terms of performance (both average and worst-case) and resources.

The longer-term goal of this research is to enable the standardization of correct rounding for elementary functions [8].

1.4 Contributions of this article

This article summaries recent advances towards this goal, supported by experimental results.

We first recall in Sec. 2 a range of techniques used for correctly rounding an elementary function in the portable `crlibm` library. Section 3 then relaxes the condition of portability to

study the impact of specific processor features (double-extended precision, fused multipply-and-add, multiple floating-point status registers), and develop techniques targetting such processors. This raises practical questions, which Section 4 tries to answer by implementing two functions (arctan and exp) on two processor families which support double-extended precision (Pentium and Itanium). These implementations of correctly rounded exp and arctan have worst-case execution times respectively less than 2× and 8× the time of the (not correctly rounded) best available vendor implementation. This is an improvement on the typical 1000× factor of `libultim`, all other things (in particular average time, code size and memory consumption) being similar or improved. The techniques are very general, and similar results could be obtained for most elementary functions. In addition, this performance improvement has an important impact on the *design cost* of writing a proven, correctly rounded implementation, which is discussed in Section 5 along with other implementation considerations.

2 The `crlibm` approach

Ziv's Ultimate Mathematical Library performs is entirely based on IEEE-754-compliant double-precision FP arithmetic: the first few steps compute an approximation to the function as a the sum of two double-precision number (which we classically call a *double-double* in the following). Subsequent, more accurate steps use a FP-based multiple-precision package which may provide up to 800 bits of precision, hence the astronomical confidence.

2.1 Tight worst cases for correct rounding

A first practical improvement over Ziv's approach derives from the availability of tight bounds on the worst-case accuracy required to compute many elementary functions, computed by Lefèvre and Muller [15] using ad-hoc algorithms. Some functions are completely covered (most notably exponential and logarithm), some are still being processed and should be covered within a few years. However, some functions (most notably the trigonometric functions and some special functions) are out of reach of current methods: there is currently no hope of computing their worst cases except on a restricted interval. However, the interval for which current algorithms work is the most practically useful (e.g. around zero for the trigonometric functions).

Knowing the worst case required accuracy for a function, it is possible to tailor Ziv's approach to match it. More specifically, `crlibm` implements only two steps of Ziv's algorithm, the second one being accurate enough to cover the worst case required accuracy. This is not only more efficient, it also makes it much easier to *prove* that an implementation actually returns the correctly rounded result. The `crlibm` distribution includes a detailed description of each implementation, including an attempt at such a proof. This proof mostly consists in computing a tight error bound on the overall error of the first step, as explained below.

2.2 Portability

The mainstream `crlibm` implementation intends to be portable to most systems, assuming only C99 compliance and the availability of IEEE-754-compliant format and operations. Therefore its first step computes a result as a double double-precision number $y_h + y_l$. Its second step uses an ad-hoc, integer-based multiple-precision library called `scslib` [7].

2.3 Rounding tests and precision/performance tradeoffs

The approximation $y_h + y_l$ computed in the first step is used to decide if the second step needs to be launched. For rounding to the nearest, the following test (also present in `libultim`) is used:

```
if(y_h == (y_h + (y_l*e)) ) return y_h;
else /* launch accurate phase */
```

Here we use the C syntax (== is the equality comparison operator), and e is a double-precision number computed out of the overal relative error $\overline{\varepsilon}$ of the first step as $e \approx 1 + 2^{54}\overline{\varepsilon}$. The exact value of e, the validity conditions of this test, and the proof that it ensures correct rounding, are detailed in the documentation of `crlibm` [1]. Similar tests are also given for directed rounding modes, they are conceptually simpler, therefore we concentrate in the sequel on round to the nearest. However it should be clear that our goal is to offer the four rounding modes.

The rounding test here depends on a constant e which is computed out of the overall relative error bound. This gives an hint at the 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`), and p_2 is the probability of launching the second phase (typically we aim at $p_2 = 1/1000$ so that the average cost of the second step is less than 1/10.

The value of e in the test implies that p_2 is almost proportional to $\overline{\varepsilon}$. Therefore, to minimise the average time, we have to

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

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 makes sense to have different rounding constant e on these different paths [17, 6].

3 Beyond `crlibm`

3.1 Modern Floating-Point Units

Most recent processors offer specific hardware features which cannnot yet be used in a portable way. For our purpose, the most significant of these features are:

- Double-extended precision with 64 bits of mantissa instead of 53 in double-precision, as specified in the IA-32 and IA-64 instruction sets (implemented by the Pentium-compatible and Itanium processors respectively). Note that the IEEE-754 standard gives a more general definition of double-extended precision, but it has not yet been

translated as a usable, standard combination of processor/compiler/system. Therefore, in the remaining of this paper, the meaning of the phrase "double extended" will be that of the IA-32 specification (which is also included in the IA-64 specification).

- Fused floating-point multiply-and-add operators (FMA), as available in the Power/PowerPC and Itanium architectures. These operators allow improvement in performance (as they perform two operations in one instruction) but also in accuracy, as only one rounding is performed. Most significant to us is that an FMA reduces the cost of Dekker algorithm (which computes the exact product of two FP numbers as the sum of two FP numbers [9, 14]) from 17 operations down to only 2, with a corresponding reduction of the cost of the double-double multiplication.

- A low overhead of changing the rounding mode or working precision, thanks to the availability of several floating-point status registers (FPSR) selected on an instruction basis. This is a feature of the Itanium processor family. Comparatively, other processors have only one FPSR, and changing it (e.g. to change the working precision) typically requires flushing the FP pipelines.

These features are being used for the standard evaluation of elementary functions [16], and some were actually designed to this purpose [19]. We now discuss their impact on the evaluation of a double-precision correctly rouded elementary function.

3.2 Correct rounding using double-extended arithmetic

Recently [5] one of us studied specifically the impact of using double-extended arithmetic for computing functions correctly rounded to double-precision.

3.2.1 First step in double-extended

A first obvious idea is to compute the first step in double-extended precision, wich removes the need for double-double arithmetic in this step. This yields some performance improvement, typically up to 50%.

However, on architectures implementing the IA-32 instruction set, this approach requires changing the rounding mode of the processor, at least when entering the function (to convert the input x to a double-extended) and when leaving it (to return a double). This takes more than 20 cycles on the Pentium-4 processor in our experiments, and takes back a lot of the interest of using double-extended precision. On the Itanium processors, however, there is no such penalty.

3.2.2 Second step in double-double-extended

The performance that double-extended precision can bring to the second step is more dramatic. This format provides 64 bits of mantissa, so that the sum of two double-extended numbers (a *double-double-extended*) will hold 128 bits of precision. Unfortunately, Muller and Lefèvre found that for some functions (including exp, cos and tan), correct rounding required an intermediate accuracy higher than 2^{-130} (up to 2^{-157} for the exponential): It is therefore not possible to compute an intermediate result to such relative accuracy as the sum of two double-extended numbers.

The central remark in [5] is that such bad cases always happen for very small values of the input number x. In such cases, a Taylor approximation provides a straightforward method for approximating the function as the sum of *three* double-extended numbers[1]: More specifically, an approximation of the function as $1+p(x)$ or $x+p(x)$, where $p(x)$ is computed as a double-double-extended $p_h + p_l$, will hold the required relative accuracy [5]. Table 1 illustrates this situation for the two functions studied in this paper.

Exploiting the fact that the most significant term of this sum (1 or x) is representable as a double-precision number, it is then possible to recover the correct rounding of $1 + p_h + p_l$ (or $x + p_h + p_l$) to double precision, using a sequence of 5 double-extended additions [5].

Function	Interval of x	WCA on f	WCA on p
$y = e^x$	$[2^{-54}, 2^{-44}]$	2^{-158}	2^{-115}
	$[2^{-44}, 2^{-30}]$	2^{-138}	2^{-109}
	$\lvert x\rvert \geq 2^{-30}$	2^{-113}	
$y = \arctan(x)$	$[\tan(2^{-25}), \tan(2^{-18})]$	2^{-126}	2^{-109}
	$[\tan(2^{-18}), 2]$	2^{-113}	

Table 1: Worst-case accuracy (WCA) required for double-precision correct rounding of exp and arctan.

3.2.3 Rounding test

The rounding test presented in 2.3 assumes double-precision arithmetic: it has to be adapted if the first step now returns a double-extended number y_{de}. The straightforward idea is to build $y_h = \text{RoundToDouble}(y_{de})$ and $y_l = y_{de} - y_h$ (which will be an exact operation), then use the test of 2.3 on y_h and y_l. When targetting the Itanium processors, the cost is altogether 4 operations and about 16 cycles, thanks to the fact that the precision and rounding mode are controlled on an operation basis, with no overhead. This cost might be reduced further.

When targetting IA-32 processors, this option could involve several costly changes of the precision (to double to compute y_h, then back to double-extended to compute y_l, then back to double to compute the test, then possibly back to double-extended for the second step). Fortunately the first conversion of y_{de} to the nearest double may be performed by the memory unit, without changing the precision of the FPU. Besides we had to perform a change of precision anyway before returning the result when the first step is enough. Therefore the following sequence minimises the number of precision changes:

- Compute $y_h = \text{RoundToDouble}(y_{de})$ by writing it to memory and reading it back
- Compute $y_l = y_{de} - y_h$ in double-extended
- Change the FPU precision to double

[1] This is not a coincidence: These worst cases are indeed highly improbable according to Gal's statistical argument [11, 22]. This argument predicts that the worst case accurracy for a correctly rounded function to double is expected at $2^{-53-64} = 2^{-117}$, and that a worst case accuracy of 2^{-157} has probability 2^{-40} of happening. The fact that such worse-than-expected cases indeed happen is a direct consequence of the availability of a Taylor approximation of the function, which breaks the asumptions of randomness in Gal's reasoning.

- perform the rounding test et return y_h if successful
- otherwise change the FPU precision back to double-extended and start second step

The only wasted time in this approach is when going for the second step, where we have two changes of precision, but this occurs rarely. However we also designed a rounding test which is closer to the intuition depicted in 1.2 and can be computed entirely in double-extended precision: To test whether $y_h + y_l$ is close to the middle of two consecutive floating-point numbers, it first computes $u = 0.5\text{ulp}(|y_h|)$, then compares $|y_l| - u$ to $|y_h| \times \overline{\varepsilon}$. It turns out to be slower in average.

3.3 Practical questions

This work left open a few practical questions relative to the implementation of a correctly rounded functions [5].

1. What is the relative performance of the second step and the first step ?
2. What implication does this have on the precision/performance tradeoff ?
3. Is it worth at all designing two steps if the second one is only twice as slow as the first one, considering the cost of the rounding test ?
4. Can we reuse intermediate results from the first step in the second one ?
5. Can we design algorithms sharing tables and intermediate values, which are efficient both for the first step and the second step ?
6. What is the uncompressible cost of correct rounding, *i.e*, the cost of the rounding test (assuming that we have taken the best possible accurate-faithful algorithm as a first step, and that the average cost of the second step is negligible) ?

The experiments described in the next section were carried out to study these questions. The last section will draw more conclusions.

4 Experiments and results

For these experiments, we chose the exponential function because it is the easiest to implement, and the most often cited in the litterature about elementary function implementation. Conversely, we chose the arctangent as being comparably expensive to implement: It is approximated by polynomials of comparatively larger degree, and its argument reduction requires either very large tables, or a division. Some other functions present specific difficulties (trigonometric and special) but they are left out of this study because their worst-case required accuracy is unknown so far

In all the tests, input random numbers were chosen in a range which avoids the (less meaningful) special cases. For these special cases, the timings may be much faster, but also much slower (up to 18,000 cycles when denormals appear and the underflow flag is raised on the Itanium 1). We compare our results to vendor `libm`s which are highly optimized and accurate-faithful.

4.1 Arctangent on the Pentium processors

Here we chose an algorithm which allows both steps to share the argument reduction, special case handling, and some intermediate computations. This algorithm is described in appendix. Tables 2 and 3 show some absolute timings, in cycles. The test conditions were as follows:

- All these timings were measured on machines running Linux, with the gcc-3.3 compiler.
- All these timings include the cost of a function call, which is about 25 cycles on both architectures. In other words, to get the time actually spent computing the function, you should subtract 25 to these numbers.
- The cost of changing the floating-point status register twice is about 40 cycles on both architectures.
- The `libultim` library is faster here in average than the vendor `libm`. The reason is that it uses a table of about 40KB to perform an argument reduction without division. The `crlibm` versions, in contrast, use less than 4KB of tables, at the expense of a division.

arctan PIII	avg time	max time
`mpfr`	384865	3231586
`libultim`	210	150651
`crlibm` portable	324	16024
`crlibm` using DE	233	1113
`crlibm` using DE, new test	247	1085
`libm` (no correct rounding)	*160*	*172*

Table 2: Timings in cycles for some implementations of a correctly-rounded arctangent on the Pentium III.

arctan P4	avg time	max time
`mpfr`	438742	3955724
`libultim`	218	267548
`crlibm` portable	441	44700
`crlibm` using DE	312	1736
`crlibm` using DE, new test	295	1904
`libm` (no correct rounding)	*242*	*332*

Table 3: Timings in cycles for some implementations of a correctly-rounded arctangent on the Pentium 4.

The increase in pipeline depth is here apparent in the fact that the P4 needs almost twice as many cycles as the PIII in the worst case for most implementations. Besides, some timings are inconsistent with expectations. In particular, the new rounding tests perform as expected

on the P4 (lower worst case but higher average case) but the situation is inverted (in a totally reproductible way) on the PIII. All this shows is that it is difficult to predict what will happen in these deeply pipelined architectures with optimizing compilers. It also gives hope that we can design processor-specific rounding tests better than the two we have here.

Here are some more general conclusions that could be drawn from this experiment.

- By returning a value before the rounding test, we can measure the incompressible time cost of correct rounding. Here it is about 35 cycles, or between 10 and 20% of the best current accurate-faithful implementation.
- The worst case time here is 7-8 times the average time of the best current accurate-faithful.
- However our first step alone is also 10-20% slower than the vendor `libm`, so there is still room for improvement in the algorithm. However it is sufficiently low to conclude that sharing tables and values between the first and second step doesn't incur a major performance penalty.

4.2 Arctangent on the Itanium

This first Itanium experiment uses the same algorithm as previously (and exactly the same 4KB tables), but tries to perform additional optimizations by using FMAs wherever possible, especially to speed up double-double arithmetic (In the first step, we also replaced the library double-extended division with a less accurate one to save a few cycles, and used a parallel implementation of the Horner recurrence). Such low-level optimizations cannot be done efficiently using current gcc, because inserting assembly instructions in C code leads to poor scheduling. Therefore we used the Intel icc8.1 compiler for Linux to compile the arctangent. This compiler support a range of intrinsics giving a high-level access to most assembly-language instructions, including FMAs. The full source code is available as file `atan-itanium.c` in the `crlibm` CVS repository. Table 4 shows some absolute timings, in cycles.

arctan Itanium-1	avg time	max time
`mpfr`	407523	3416213
`libultim`	277	211306
`crlibm` portable	363	8453
`crlibm` using DE, two steps	136	690
`libm` (no correct rounding)	*107*	*112*

Table 4: Arctangent timings in cycles on an Itanium-1 processor.

Here are some comments on these tables:

- These numbers include the cost of a function call, which is about 37 cycles.
- The improvement over crlibm-portable is more dramatic than for the Pentium, because of the FMA, and because using double-extended doesn't incur the cost of flushing the

pipeline on the Itanium as already explained. This is also the reason why we use a rounding test *à la* Ziv.

- Here we measured that the uncompressible cost of the correct rounding test is about 16 cycles. It is therefore the main contribution to the average overhead of correct rounding. Again it is about 15% of the best current accurate-faithful implementation.
- The worst case time here is again 7 times the average time of the best current accurate-faithful. This is somehow disappointing, because the FMA, speeding up double-double multiplication by a factor 8, should bring a proportionnally larger improvement to the second step than to the first step. We believe their is room for improvement here. Note for instance that Hewlet Packard's Markstein [19] described a quad-precision arctangent in HP-UX accurate to 0.5001 ulp (which should be enough to derive a second step) within 321 cycles[19]. It uses a sequence of only 5 FMAs for computing one double-double Horner steps, where our code uses standard double-double algorithms amounting to 15 FMAs. The error bound for this sequence "still requires formal proof" [19], but it is an obvious – and very generally applicable – optimization direction to research further.

4.3 Exponential on the Itanium

In this experiment, a second step was first derived from an Intel quad-precision routine, and then a first step was derived from the second step, using the same range reduction. An overview of the algorithm used is given in appendix. The full source code is available in the `crlibm` CVS repository from `http://lipforge.ens-lyon.fr/projects/crlibm/`, file `exp-itanium.c`.

We timed the first step alone (to check it matches the performance of the standard `libm`), the second step alone (since it makes a self-sufficient correctly-rounded exp), and the two-step algorithm. We also timed the Linux standard `libm` (derived from an older version of Intel open-source optimized `libm`), and the portable version in `crlibm`. Table 5 shows some absolute timings, in cycles.

exp Itanium-1	avg time	max time
`libultim`	193	2439385
`mpfr`	24540	115152
`crlibm` portable	295	5633
`crlibm` using DE, two steps	100	162
`crlibm-DE`, second step alone	124	126
`libm` (no correct rounding)	*89*	*89*

Table 5: Exp timings in cycles on an Itanium-1 processor.

Here are some comments on this table:

- Here ve have a relatively table-hungry algorithm (8KB). It is one of the reasons why our first step, if we remove the 16 cycles of rounding test, is faster than the `libm` (which was written in assembly by Intel people, but only uses 1KB of tables).

- Adding the first step to the code of the second step meant adding 13 lines only, since we reuse the special cases handling and the range reduction. These 13 lines take 160-126=34 cycles for
  - a Horner evaluation,
  - a reconstruction,
  - and the rounding test which is statically predicted not to go for the second step (and does in this case).
- Here it may not be worth having a two-step algorithm. Or, if we do write a two-step algorithm, it will be one which has a slower second step to use a smaller table. For instance we are confident that a 4KB exp can be written with a less-than-10x worst case.

5 Designing the post-ultimate libm

The immediate conclusion of the previous section is that using double-extended arithmetic, it is possible to design a correctly-rounded function whose worst case is within 10x of the best accurate-faithful, and whose average performance is only degraded by the incompressible cost of the rounding test.

We now discuss the effort involved in designing such a function.

5.1 Reuse and share

The first idea is to reuse existing, well optimized algorithms. This is obvious for the first step, which can be derived from a standard `libm` implementation as soon as this implementation is faithful. Recent vendor libraries now return the correct result for more than 95% of the inputs [12, 16]. Deriving a first step from such a function involves

- adding a rounding test, which is easy, and
- proving a tight bound on the overall error, which may be difficult (of course, the more clever and sophisticated the algorithm, the more difficult the proof).

Recently, Andrey Naraikin and Aleksey Ershov (from Intel Nizhniy Novgorod Lab) suggested that the second step could be similarly derived from a quad-precision implementation. Here, quad-precision usually means a 128-bit format with 112 bits of mantissa, and the functions are computed with a higher relative accuracy to keep the error very close to an half-ulp [19]. When double-extended precision is available, this is done thanks to double-double-extended techniques.

As a conclusion, should a vendor commits itself to correctly-rounded double-precision functions, a lot of the work would be shared between double-precision and first step (at least the handling of exceptional cases), and between quad-precision and second step. Having this sharing in mind from the beginning will allow algorithms which are as efficient as current accurate-faithful algorithms, as our amateur arctangent suggests. Again, the real design cost would be in proving error bounds systematically.

5.2 Making proofs easy

Now we discuss the design cost of computing the error bounds. For the second step, it makes sense to have a large overkill of accuracy in the algorithm if it makes the proof simpler (typically having a coarse estimate on each rounding error): the average performance impact will be negligible, and the worst-case impact remains acceptable. In `crlibm` for instance, our ad-hoc multiple-precision library is much too accurate (to 200 bits) because it gives much freedom in designing the algorithms and proving them. More specifically, it allows to concentrate on approximation errors, because coarse bounds on rounding errors will suffice. This is still the case for a double-double-extended second step, because the actual worst cases accuracies required are about 2^{-117}, and rounding to double-double-extended entails error smaller than 2^{-128}.

Now for the first step, we have to minimise p_2 and this means computing a tight bound on the error. However, an important consequence of the 10x factor and of a double-extended first step is that we may now be much lazier in this computation, and for the same reason:

- The second step will now be within a factor 10 of the first step, so $T_2 < 10T_1$.
- A coarse computation of the rounding errors in the double-extended first step will typically sum up to a term smaller than $\overline{\varepsilon} = 2^{-63}$, which would translate to $p_2 \approx 1/1000$. The contribution of this lazy error bound to the average time is therefore about $T_1/100$, which is negligible. Therefore, here also, we may concentrate on the approximation errors, which are simpler to manage.

We therefore find out that the cost of implementing a correctly rounded function using double-extended arithmetic is much reduced when compared to the cost in portable crlibm, where we had to

- compute a tight error bound on the first step, because being lazy had an impact on performance, and
- write the second step using `scslib` and its proof, sharing only little of this work with the first step.

As a final remark, it may even happen that a correctly rounded implementation provides faster average performance than an accurate-faithful implementation. The idea is here to have a first step which is deliberately less accurate thant the accurate-faithful version, and hence faster enough to compensate for the cost of the rounding test.

6 Conclusions

It is known since Ziv's work that it is possible to write elementary functions which are correctly rounded with astronomical probability, with a very small average performance overhead over the current best implementation.

This paper shows, with experimental support, that double-extended arithmetic allows to write functions which are proven correctly rounded to double precision, with a worst case overhead of less than a factor ten, and with predictible and acceptable memory consumption. It also explains how to write such efficient correctly rounded functions with little effort.

We believe that those overheads are comparable to those imposed on the hardware by IEEE-754 compliance, and that the benefits of correct rounding are worth this minor performance loss, as it was for the four operations.

Our aim is now to see a gradual generalisation of correctly-rounded functions in mainstream systems (currently, only Linux incorporates a derivative of Ziv's library, and it is actually enabled only if double-precision is the default in the system [2]). For the functions for which the worst-case accuracy required is known (most notably the exp and log family), there is no longer any technical obstacle preventing this generalisation. For other functions (most notably the trigonometric functions), we will offer proven correct rounding on a small interval only, and on the rest of the function's range, astronomical confidence only. Such a multilevel approach may even be formalised as a standard [8].

In addition to actually writing complete post-ultimate `libm`s, there are several research directions to explore.

- The proof framework of `crlibm` needs to be improved. Currently, a proof is a mixture of source code, LaTeX and Maple which provides an extensive and open documentation of each function, but doesn't necessarily inspire confidence as a proof.
- Generic support for FMAs and double-extended precision with compile-time macros could also be added to `crlibm`. Here the difficulty is to manage the error computation in the combination of possible cases.
- For processors without double-extended support (and for computing double-extended correctly-rounded functions), a combination of accurate tables [11] and a limited amount of triple-double computation should be explored.

Acknowledgements

Many thanks go the people of Intel Nizhniy Novgorod Lab for interesting discussions, and for giving us access to some of their code. Special thanks go to Andrei Naraikin and Sergei Maidanov, and to Christoph Lauter and Aleksey Ershov for their work on the exponential. The `crlibm` project was partially funded by INRIA, France.

References

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

[2] Test of mathematical functions of the standard C library. http://www.vinc17.org/research/testlibm/index.en.html.

[3] ANSI/IEEE. Standard 754-1985 for binary floating-point arithmetic, 1985.

[4] Marc Daumas and Claire Moreau-Finot. Exponential: implementation trade-offs for hundred bit precision. In *Real Numbers and Computers*, pages 61–74, Dagstuhl, Germany, 2000.

[5] 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. Available at http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2004/RR2004-10.pdf.

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

[7] D. Defour and F. de Dinechin. Software carry-save for fast multiple-precision algorithms. In *35th International Congress of Mathematical Software*, Beijing, China, 2002. Updated version of LIP research report 2002-08.

[8] D. Defour, G. Hanrot, V. Lefèvre, J.-M. Muller, N. Revol, and P. Zimmermann. Proposal for a standardization of mathematical functions. *Numerical algorihms*, 2004.

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

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

[11] 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.

[12] J. Harrison, T. Kubaska, S. Story, and P.T.P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, Q4, 1999.

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

[14] D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, MA, 1973.

[15] 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.

[16] R.-C. Li, P. Markstein, J. P. Okada, and J. W. Thomas. The libm library and floating-point arithmetic for HP-UX on Itanium. Technical report, Hewlett-Packard company, april 2001.

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

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

[19] Peter Markstein. A fast quad precision elementary function library for Itanium. In *Real Numbers and Computers*, pages 5–12, Lyon, France, 2003.

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

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

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

[23] P. T. P. Tang. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 15(2):144–157, June 1989.

[24] 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, Los Alamitos, CA.

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

A Overview of the exponential

Both steps use the following equation, inspired by an algorithm in [23].

$$e^x \approx 2^M \cdot 2^{\frac{i_1}{2^7}} \cdot 2^{\frac{i_2}{2^{14}}} \cdot (1 + p\,(x_h)) \cdot (1 + x_l) \tag{2}$$

The argument $x \in [-745.13; 709.78]$ (the range in which e^x is representable in double-precision and not rounded to zero) is decomposed as the sum of three terms:

$$x \approx k \cdot \frac{\ln 2}{2^{14}} + x_h + x_l$$

where x_h and x_l are two double-extended numbers such that $|x_h| < \frac{\log 2}{2^{15}}$, $|x_l| < 2^{-64}|x_h|$, and

$$k = \left\lfloor x \cdot \frac{2^{14}}{\ln 2} \right\rceil$$

The integer k is itself decomposed into three terms:

$$k = 2^{14} \cdot M + 2^7 \cdot i_1 + i_2 \tag{3}$$

where $i_1, i_2 \in \mathbb{N} \cap \left[0; 2^7 - 1\right]$ and $M \in \mathbb{N}$.

The terms $2^{\frac{i_1}{2^7}}$ and $2^{\frac{i_2}{2^{14}}}$ are precomputed and stored in two 128-entry tables. Each entry consists of two double-extended numbers providing 128 bits of precision: $t_{h1} + t_{l1} \approx 2^{\frac{i_1}{2^7}}$ and $t_{h2} + t_{l2} \approx 2^{\frac{i_2}{2^{14}}}$. The first step only uses t_{h1} and t_{h2}.

The polynomial $p(x_h)$ approximates the exponential of the reduced argument as a modified Remez polynomial [22] of degree 4 in the first step, and degree 6 in the second step (using two double-double-extended FMAs in the latter case). The reconstruction consists in computing the product of all the terms of Eq. 2.

B Overview of the arctangent

A fully detailed description is available as part of the `crlibm` documentation [1]. We compute $\arctan(x)$ as

$$\arctan(x) = \arctan(b_i) + \arctan(\frac{x - b_i}{1 + x.b_i}) \tag{4}$$

The b_i are exact doubles and the $\arctan(b_i)$ are stored in double-double. We define $X_{\text{red}} = \frac{x - b_i}{1 + x.b_i}$.

We tabulate intervals bounds a_i and values b_i such that

$$x \in [a_i; a_{i+1}] \Rightarrow \frac{x - b_i}{1 + x.b_i} < e \quad . \tag{5}$$

The i such that $x \in [a_i; a_{i+1}]$ will be found by dichotomy. Therefore we choose a power of two for the number of intervals: 64 intervals ensure $e = 2^{-6.3}$.

Then $\arctan(X_{\text{red}})$ is approximated by a polynomial of degree 9 in the first step, and degree 19 in the second step.

Finally, the reconstruction implements equation (4) in double-double arithmetic.