Correct Rounding in Double Extended Precision

Sélène Corbineau
Département d'Informatique de l'ENS
École Normale Supérieure-PSL
F-75005 Paris, France
selene.corbineau@ens.fr

Paul Zimmermann
Université de Lorraine
CNRS, Inria, LORIA
F-54000 Nancy, France
paul.zimmermann@inria.fr

***Abstract*—The double extended precision format is an 80-bit floating-point format introduced in the 80x87 series of floating-point processors by Intel. Since the introduction of vector instructions in the x86 processors, its use has fallen due to speed concerns. We implement in CORE-MATH the first correctly-rounded routines for double extended precision. These implementations use modern microprocessor features and double-double arithmetic, avoiding x87-specific features, and achieve up to 2x speedup over state-of-the-art implementations which are not correctly rounded. This demonstrates that double extended precision could be viable as a large computational format.**

***Index Terms*—IEEE 754, floating-point, correct rounding, double extended format, efficiency.**

I. Introduction

The double extended precision format was introduced in Intel's x87 series of floating-point coprocessors. This floating-point format has a 80-bit encoding: a sign bit, an exponent encoded on 15 bits, and a significand on 64 bits (with no implicit leading bit). Featuring excess precision and range compared to the double precision format (now binary64), it allowed double precision intermediate computations to be carried out without excessive accumulation of errors. Except for some minor encoding quirks, it can be thought of as the natural extension of the IEEE 754 standard's binary floating-point formats to 80 bits. Most C compilers for the x86 and x86-64 architectures assign double extended precision to the `long double` type. Intel later introduced vector instructions in the x86 architecture to assist multimedia and signal processing applications. These SIMD extensions offer at most double precision, but are faster and easier to compile for. They proved popular and usage of double extended precision dwindled.

Current `long double` mathematical functions implementations perform all their calculations in the slow native double extended format. Most are a thin wrapper around x87 complex mathematical instructions such as FPSIN or FPEXP2 which are microcoded, unmaintained, and yield processor-dependent results. These instructions are slow even compared to naive computation in the double extended format and their accuracy has been the object of errata.

A solution against the reproducibility problem is to enforce usage of correctly-rounded routines. However, prior to this work, the major correctly-rounded libraries (CRLIBM, MathLib, LLVM-libm, RLIBM, CORE-MATH) did not offer any extended double precision support. This is partly because correctly-rounded routines must compute their result to a higher intermediate precision. Except for the single precision format, where using double precision for internal computations is quite efficient, this is usually achieved by double-word arithmetic. This technique gives best performance when fused multiply-add (FMA) is available in hardware; however since this is not the case for double extended precision on current processors, it would yield unacceptably slow routines.

Leveraging the fact that double-double arithmetic is more precise than the double extended format, we implement correctly-rounded routines using double-double arithmetic in the critical path. This nearly eliminates x87 usage, allowing a significant performance boost and could be ported to allow double extended precision computation on non-x86 architectures.

Another difficulty implementing correctly-rounded routines is that certifying them traditionally involves looking for the hardest-to-round cases. This is hard for double extended precision routines because of the large input space. In this work, we show that this search can be replaced by a proof that all cases are sufficiently easy. This allows us to use more restrictive parameters in our search tools and speeds up correctness checking.

The contributions of this article are the following: after some useful routines in §II, we present in §III routines to convert a double extended number to a double-double representation, to convert a double-double representation to a 128-bit floating-point number, and to round such a 128-bit number to a double extended number. These routines are building blocks independent from the function to be implemented. Then in §IV we present a case study for the exponential function, correctly rounded in double extended precision. Finally, we present experimental results and conclude in §V.

II. Background

A. Correct rounding

For a mathematical function f and a floating-point number x, the correct rounding is the floating-point number y which is closest to $f(x)$ according to the given rounding mode. Correct rounding is a way to ensure portability of programs and reproducibility of results. For basic arithmetic operations, IEEE 754 requires correct rounding. For mathematical functions, many people believe that correct rounding is expensive, but Ziv demonstrated already in 1991—with the

2576-2265/25/$31.00 ©2025 IEEE
DOI 10.1109/ARITH64983.2025.00027

MathLib library—that it can be computed efficiently [11]. For more details on correct rounding, we refer the reader to [1].

B. Double-double arithmetic

A "double-double" number is an evaluated sum $h + \ell$ of two double-precision numbers. If $h = \circ(h + \ell)$ with the rounding mode $\circ(\cdot)$, we say the double-double number is "normalized". Normalizing double-double numbers after each operation simplifies the error analysis, but is expensive: we usually try to avoid it, still trying to keep a rigorous bound on the overlap between h and ℓ. For more details on double-double arithmetic, see [9].

We recall here three main algorithms on double-double arithmetics: Algorithm ExactMul (resp. FastTwoSum) produces a double-double number for the product (resp. sum) of two double-precision numbers (we use here the formulation from [4]), and Algorithm DMul approximates the product of two double-double numbers. In the absence of underflow and overflow, ExactMul is always exact, whatever the rounding mode. In ExactMul, we denote by $\circ(ab - h)$ the correct rounding of $ab - h$, which can be performed with a fused multiply-add.

Algorithm 1 (ExactMul)

Input: $a, b \in \mathbb{F}$
Output: h, ℓ such that $h + \ell = ab$
$h \leftarrow \circ(ab)$
$\ell \leftarrow \circ(ab - h)$

Algorithm 2 (DMul)

Input: $a_h, a_\ell, b_h, b_\ell \in \mathbb{F}$
Output: x_h, x_ℓ approximating $(a_h + a_\ell)(b_h + b_\ell)$
1: $(x_h, s) \leftarrow \mathrm{ExactMul}(a_h, b_h)$
2: $t \leftarrow \circ(s + a_\ell b_h)$
3: $x_\ell \leftarrow \circ(t + a_h b_\ell)$

We will make extensive use of the following lemma.

Lemma 1: Using the notations of Algorithm 2, let $A_h, A_\ell, B_h, B_\ell \in \mathbb{R}_+$ be bounds on $|a_h|, |a_\ell|, |b_h|, |b_\ell|$ respectively. Let RU be the rounding toward infinity. Let us call

$$T = \mathrm{RU}(\mathrm{ulp}(\mathrm{RU}(A_h B_h)) + A_\ell B_h), X_\ell = \mathrm{RU}(T + A_h B_\ell).$$

Then, assuming no underflow nor overflow, the error of DMUL is at most $\mathrm{ulp}(T) + \mathrm{ulp}(X_\ell) + A_\ell B_\ell$, and X_ℓ is such that $|x_\ell| \leq X_\ell$.
Furthermore, we also have $|x_h| \leq X_h = \mathrm{RU}(A_h B_h)$.

Proof: We have $|x_h| \leq \mathrm{RU}(A_h B_h)$ given EXACTMUL. Then we see that $|s| < \mathrm{ulp}(x_h)$, so that $|s| < \mathrm{ulp}(\mathrm{RU}(A_h B_h))$. Given line 2 we get that $|t| \leq T$. The rounding error on line 2 is therefore at most $\mathrm{ulp}(T)$. In the same vein, we get that $|x_\ell| \leq \mathrm{RU}(T + A_h B_\ell) = X_\ell$ so that the rounding error on line 3 is at most $\mathrm{ulp}(X_\ell)$. Since we neglected the term $a_\ell b_\ell$ from the whole product, we get the claimed error bound. ■

Still in the absence of underflow and overflow, FastTwoSum is exact for rounding to nearest, but might not be for directed rounding modes (see [2], [6] for more details).

Algorithm 3 (FastTwoSum)

Input: $a, b \in \mathbb{F}$ with $a = 0$ or $|a| \geq |b|$
Output: h, ℓ such that $h + \ell$ approximates $a + b$
$h \leftarrow \circ(a + b)$
$t \leftarrow \circ(h - a)$
$\ell \leftarrow \circ(b - t)$

C. The double extended format

The double extended format is an 80-bit format, with a 1-bit sign s, a 15-bit exponent e, and a 64-bit significand m. Apart from special values (NaN, Inf) and subnormal numbers, the corresponding value is $(-1)^s \cdot 2^{e-16383} \cdot m$, where m is interpreted as a number in $[1, 2)$, and always has it most significant bit set (contrary to the binary32 and binary64 formats, the leading bit is explicit). This format has 11 bits more precision than binary64, and a larger exponent range, since it can represent numbers as large as about 2^{16384}, and as small as 2^{-16445}. This format usually corresponds to the C type `long double` on x86 and x86-64 processors. In the rest of this article, since we target such processors, both "double extended" and "long double" refer to that format.

D. Notations

We write $\mathbb{Z}_p$ the set of p-bit numbers in two's complement notation. We denote $\mathbb{F}_p$ the set of p-bit floating-point numbers, and DD the set of double-double numbers. For $x \in \mathbb{F}_p$, we denote by $\mathrm{ulp}(x)$ the unit-in-last place of x [8]. We also denote $\mathrm{ulp}_q(x)$ the unit-in-last place of x, considered as a q-bit number (with unbounded exponent).

III. EVALUATION STRATEGY

Since our aim is to use only double-double arithmetic in the critical path, we present here routines to convert between the extended double and double-double representations. These routines are independent from the function to be evaluated.

A. Conversion from double extended to double-double

The double-double representation has 11 bits for encoding the exponent, which does not allow representing all double extended values. Very large and very small inputs thus have to be handled separately. However, this is usually the first reduction step of most algorithms computing mathematical functions anyway. Depending on the function, we discard inputs or scale them directly using the exponent as stored on the stack prior to the routine call. This avoids an x87 `FXSCALE` instruction, replacing it by a few integer instructions.

Assuming we reduced to a long double input in the double precision exponent range, we use the following algorithm:

Algorithm 4 Long double splitting (LDSPLIT)

Input: $x \in \mathbb{F}_{80}$
Output: $(a, b) \in \mathbb{F}_{64}^2$ such that $|b| < \mathrm{ulp}(a)$ and $a + b = x$
$a \leftarrow \circ_{64}(x)$
$b \leftarrow \circ_{64}(x - \mathbb{F}_{80}(a))$

Lemma 2: If $|x| \leq 2^{1024}(1 - 2^{-53})$ and $|x| \geq 2^{-1011}$ then Algorithm 4 is correct.

Proof: The condition $|x| \leq 2^{1024}(1-2^{-53})$ ensures there is no overflow computing a. Then we know that $|x - a| < \mathrm{ulp}(a)$. Let m be a's exponent and n be x's exponent. Then $m = n$ or $m = n+1$. This implies that $\mathrm{ulp}(a) = 2^{64-53+m-n}\,\mathrm{ulp}(x) \leq 2^{12}\,\mathrm{ulp}(x)$. Moreover, since a and x are integer multiples of $\mathrm{ulp}(x)$, so is $x-a$. Therefore $x-a = k \cdot \mathrm{ulp}(x)$ for some $k \in [-2^{12}, 2^{12}]$. Since $|x| \geq 2^{-1011}$, $\mathrm{ulp}(x) \geq 2^{-1074}$. Together with $|x-a| \leq |x|$ this ensures that computing b is exact, thus the correctness of the algorithm. ∎

Except for a final `FLD` due to System V's ABI, this algorithm constitutes our only use of x87 instructions in the critical path. It compiles to a `FLDT,FSTL,FSUBL,FSTPL` dependency chain. In our implementations, integer reduction computations using x directly from the stack partially hide the latency.

B. Reconstruction

Near the end of the critical path, we get a double-double approximation of $f(x)$ as $a+b$ with $|b| < \mathrm{ulp}(a)$. We convert (a, b) to an intermediate representation q whose significand has 128 bits, then round q to a long double, checking whether our error bounds ensure this result is correct in the process. To convert (a, b) to q, we use Algorithm 5.

Algorithm 5 Computing a 128-bit approximation q

Input: $a, b \in \mathbb{F}_{64}, a = (-1)^{a_s}(2^{53}+a_m)2^{a_e}$ with $a_s \in \{0,1\}$ and $0 \leq a_m < 2^{52}$, similarly for b

Output: $q = (a_s, q_e, q_m) \in \mathbb{F}_{128}$ representing $(-1)^{a_s}2^{s_e-127} \cdot q_m$ and approximating $a+b$

1: $q_e \leftarrow a_e$

2: $q_m \leftarrow 2^{127} + 2^{64+11}a_m + (-1)^{b_s-a_s}(2^{63} + 2^{11}b_m)2^{64+b_e-a_e}$

3: **if** $q_m < 2^{127}$ **then**

4: $\quad q_e \leftarrow q_e - 1$

5: $\quad q_m \leftarrow 2q_m$

return $q = (a_s, q_e, q_m)$

In line 2, the result is rounded to a 128-bit integer by truncation (if $-63 \leq t := 64 + b_e - a_e < 0$, we shift $2^{63} + 2^{11}b_m$ by $-t$ bits to the right, and if $t \leq -64$, the contribution of b is simply neglected). Notice that since $|b| < \mathrm{ulp}(a)$, we do not have to handle overflows.

Lemma 3: Assume a and b are normal double precision numbers such that $|b| < \mathrm{ulp}(a)$. Then Algorithm 5 returns q such that $q = a + b + \epsilon$ with $|\epsilon| \leq 2^{-126}|q|$.

Proof: Assume a and b have same sign ($b_s = a_s$). Then, since $|b| < \mathrm{ulp}(a)$, by summing the (scaled) significands of a and b with infinite precision, the most significant bit stays that of a. This remains true in our 128-bit format because we truncate the expansion. Then, the branch cannot be taken and there cannot be an overflow computing q_m. The correctness is then straightforward.

Assume a and b have different signs ($b_s \neq a_s$). Since $|b| < \mathrm{ulp}(a) \leq |a|/2$, $q_m \geq 2^{-1}(2^{127} + 2^{64+11}a_m) \geq 2^{-126}$. If the branch is not taken, the correctness is obvious. If the branch is taken, our previous inequality shows that we only need to shift q_e by 1 to obtain a normalized 128-bit representation. The operation in the branch does not alter $q_m \cdot 2^{s_e-127}$ which proves correctness.

The truncated part of b (if any) is less than 1, and is multiplied by 2 when the branch is taken, thus contributes to less than 2 compared to $q_m \geq 2^{127}$ at the end, which proves the bound on ϵ. ∎

Once we obtain q, we round it to a long double according to the current rounding mode. We also need to get the distance to the nearest rounding boundary. We do the following, assuming q is not in the subnormal range:

Algorithm 6 Rounding a 128-bit number q to long double

Input: $q = (q_s, q_e, q_m)$, r a rounding mode

Output: $(y, a) \in \mathbb{F}_{64} \times \mathbb{Z}_{64}$ where y rounds q and a is a scaled rounding error.

1: write $q_m = m_h 2^{64} + m_\ell$ with $0 \leq m_h, m_\ell < 2^{64}$

2: **if** $r =$ NEAREST **then**

3: $\quad m_\ell \leftarrow m_\ell + 2^{63} \pmod{2^{64}}$

4: $a \leftarrow m_\ell \,\mathrm{cmod}\, 2^{64}$ $\quad\triangleright\ -2^{63} \leq a < 2^{63}$

5: **if** $C_r(m_h, m_\ell, q_s)$ **then**

6: $\quad m_h \leftarrow m_h + 1$

7: $\quad$**if** $m_h = 2^{64}$ **then**

8: $\quad\quad q_e \leftarrow q_e + 1$

9: $\quad\quad m_h \leftarrow 2^{63}$

10: $\quad\quad a \leftarrow a/2$ $\quad\triangleright$ rounded towards zero

11: **if** $q_e \geq 16384$ **then**

12: $\quad$**return** $((-1)^{q_s}\infty, a)$

13: $y \leftarrow \mathbb{F}_{64}(q_s, q_e, m_h)$

14: **return** (y, a)

Here line 4 reinterprets the bits of m_ℓ as representing a signed number and C_r is a predicate deciding whether (m_h, m_ℓ, q_s) has to be rounded away from zero in the current rounding mode. For directed roundings, this only depends on q_s, whereas for rounding to nearest m_ℓ also plays a role.

Lemma 4: Assuming $q_e \geq -16383$ at the beginning of Algorithm 6, let (y, a) be its output. Let also $u = a \cdot 2^{-64}\,\mathrm{ulp}_{64}(y)$. Then y is the rounding of q in the current rounding mode and u is the signed distance between s and the nearest rounding boundary, rounded toward zero.

Proof: The rounding procedure is mostly straightforward, given that we exclude the denormal range. It remains to show that u is what is claimed. First, assume that there is no overflow in line 6. We show that u is exactly the signed distance between q and the nearest rounding boundary.

Assume r is a directed rounding mode. Then when q takes all possible values between two adjacent 64-bit precision numbers z^- and z^+, m_h is fixed and m_ℓ linearly increases between 0 and $2^{64} - 1$. When considered as a signed number, a goes between 0 and $2^{63} - 1$ when $z^- \leq q < (z^+ + z^-)/2$ and between -2^{63} and -1 when $(z^+ + z^-)/2 \leq q < z^+$. It is then easy to check that u is appropriately scaled such that it is as claimed.

If r is rounding to nearest, take z^- and z^+ as before. Then, when $z^- \le q \le (z^- + z^+)/2$ we see that a goes from -2^{63} to 0. Similarly, when $(z^- + z^+)/2 < q < z^+$ then a goes from 1 to $2^{63} - 1$. A similar argument to above allows us to finish the proof when there is no overflow.

The same argument can be adapted to the case where there is an overflow in line 6: the division by 2 of a explains the rounding toward zero in the theorem. ■

The reconstruction phase (Algorithm 6) is a bottleneck for two reasons:

- The branch in line 5 of Algorithm 6 is not easily predictable. To check the current rounding mode in order to compute C_r we use floating-point operations instead of directly checking the relevant CSRs.
- The System V ABI imposes that long double return values are passed in a x87 register. Since our computations are done on the stack, this imposes an `FLDT` instruction which is the hottest instruction of our routines.

IV. Case study: `expl`

We implemented several functions with correct rounding using the above mentionned techniques. We illustrate here the usage of our techniques on the example of the `expl` function (exponential in double extended format).

We use a truncated Ziv iteration method, with two precision levels: a fast path using double-double arithmetic and an accurate path using a 192-bit software ad-hoc format using 64-bit integer arithmetic. The hard-to-round cases were searched using BaCSeL (see §IV-C).

A. Fast path

The overall evaluation strategy consists in evaluating $e^x = 2^{x/\log 2}$. The input is first filtered: inputs with $|x| \ge 2^{14}$ round either to infinity, to the largest long double number, to $+0$, or to the smallest positive subnormal number depending on rounding mode and sign. By directly checking the exponent's representation, this test also catches infinities and NaNs in the input and correctly deals with them. Similarly, inputs with $|x| < 2^{-64}$ round trivially.

After this filter, all inputs easily fit in the double exponent range. We use Algorithm 4 to split the input x into (x_h, x_ℓ) exactly. All further computations proceed with double-double arithmetic using the following pseudocode.

Algorithm 7 Computing `expl`

Input: $x \in \mathbb{F}_{64}, 2^{-64} \le |x| < 2^{14}$
  $x' \leftarrow \textsc{Reduction}(x)$
  $(n, f, y) \leftarrow \textsc{Split}(x')$
  $z \leftarrow \textsc{Lookup}(f)$
  $t \leftarrow \textsc{Poly}(y)$
  $s \leftarrow \textsc{DMul}(t, z)$
  return $\textsc{Reconstruct}(n, s)$

Here, Split is a subroutine such that $\textsc{Split}(x')$ returns $(n, f, y) \in \mathbb{Z}^2 \times \mathrm{DD}$ satisfying $0 \le f < 2^{20}$ and such that $n + f/2^{20} + y$ approximates x' and $|y| \le 2^{-19.999}$. Subroutine Lookup computes an approximation of $2^{f/2^{20}}$. Subroutine Poly is a straightforward polynomial approximation of 2^y. Finally, Reconstruct is a small modification of the reconstruction phase (§III-B) to correctly round $2^n \cdot s$ to a long double. It also implements the fast path validity check knowing Algorithm 7's error bounds. By isolating the approximate exponent of the result in n we will see that quantities z, t, f, s comfortably sit inside the representable range of double-double numbers. Note that Lookup and Poly are independent and can be efficiently pipelined together.

a) Subroutine Reduction*:* We use the following algorithm, where constants K_h, K_ℓ are binary64 numbers.

Algorithm 8 Divide by $\log 2$ (Reduction)

Input: : $x \in \mathbb{F}_{64}$ with $|x| < 2^{14}$
Output: : $x' = (x'_h, x'_\ell)$ approximating $x/\log 2$
1: $(x_h, x_\ell) \leftarrow \textsc{LDSplit}(x)$
2: $K_h =$ `0x1.71547652b82fep+0`
3: $K_\ell =$ `0x1.777d0ffda0d24p-56`
4: $(x'_h, x'_\ell) \leftarrow \textsc{DMul}(K_h, K_\ell, x_h, x_\ell)$

Lemma 5: The double-double number $x' = x'_h + x'_\ell$ returned by Algorithm 8 satisfies $|x'_h| < 2^{14.7}$, $|x'_\ell| < 2^{-37.141}$ and

$$|x' - x/\log 2| < 2^{-88.951}.$$

Moreover, if the value x'_h computed by Algorithm 8 satisfies $|x'_h| < 2^{-20}$, then $|x'_\ell| < 2^{-71.63}$ and

$$|x' - x/\log 2| < 2^{-123.3}.$$

Proof: From Algorithm 4 we know that $|x_h| \le 2^{14}$ and $|x_\ell| < \mathrm{ulp}(x) \le 2^{-39}$. The constants K_h and K_ℓ satisfy $|K_h + K_\ell - 1/\log 2| < 2^{-109.5}$. Directly applying Lemma 1 (and adapting the notations), we get

$$\begin{cases} T = \mathrm{RU}(\mathrm{ulp}(K_h 2^{14}) + K_\ell 2^{14}) \le 2^{-37.873} \\ X'_\ell \le \mathrm{RU}(T + K_h \cdot 2^{-39}) \le 2^{-37.141} \\ |x'_h| \le \mathrm{RU}(K_h 2^{14}) < 2^{14.7}. \end{cases}$$

Since $\mathrm{ulp}(2^{-38}) = 2^{-90}$, the error of the DMul call is at most $2 \cdot 2^{-90} + K_\ell \cdot 2^{-39}$. The error due to $K_h + K_\ell$ not being $1/\log 2$ is at most $|K_h + K_\ell - 1/\log 2| \cdot |x| \le 2^{14-109.5}$. The total error of the algorithm is thus at most

$$2^{-89} + K_\ell \cdot 2^{-39} + 2^{-95.5} \le 2^{-88.951}.$$

If $|x'_h| < 2^{-20}$, necessarily $|x_h| \le t_0$ where $t_0 =$ `0x1.62e42fefa39efp-21`, because $x'_h = \circ(K_h x_h)$, and if we multiply K_h by $t_0 + \mathrm{ulp}(t_0)$ with rounding towards zero, we get 2^{-20}. This implies $|x_\ell| < \mathrm{ulp}(t_0) = 2^{-73}$. Using Lemma 1 with the improved bounds, we get

$$\begin{cases} T = \mathrm{RU}(\mathrm{ulp}(\mathrm{RU}(K_h t_0)) + K_\ell t_0) \le 2^{-72.827} \\ X'_\ell \le \mathrm{RU}(T + K_h \cdot 2^{-73}) \le 2^{-71.638}. \end{cases}$$

This allows us to show that the total error is at most $2^{-123.3}$ in the same way as above. ■

b) Subroutine SPLIT*:* We use a variant of the absolute splitting of [5] to truncate x' to a fixed precision 2^{-20}. We use the "magic" double precision constant $C = 3 \cdot 2^{31}$.

Algorithm 9 Splitting to 2^{-20} precision

Input: : $x' = (x'_h, x'_\ell)$ with $|x'_h| < 2^{14.7}$ and $|x'_\ell| < 2^{-37.141}$
Output: : (n, f, y) such that $n + f/2^{20} + y$ approximates x'

1: **if** $|x'_h| < 2^{-20}$ **then**
2: **return** $(0, 0, x')$
3: **else**
4: $S \leftarrow \circ(C + x'_h)$
5: extract (n, f) from S bitwise
6: $r \leftarrow \circ(x'_h - \circ(S - C))$
7: $(y_h, y_\ell) = \text{FASTTWOSUM}(r, x'_\ell)$
8: **return** $(n, f, (y_h, y_\ell))$

In line 5, f is formed as an unsigned integer by the 20 low bits of S and n is extracted from the 16 following upper bits interpreted as a two's complement integer.

Lemma 6: Algorithm 9 returns (n, f, y) such that

$$x' = n + \frac{f}{2^{20}} + y + \epsilon$$

with n, f integers, $0 < f < 2^{20}$, $|y|, |y_h| < 2^{-19.999}$, $|y_\ell| \leq 2^{-71.63}$ and $|\epsilon| < 2^{-87.922}$.

Proof: If $|x'_h| < 2^{-20}$, we have $y_h = x'_h$ and $y_\ell = x'_\ell$, and the statement follows from Lemma 5, with $\epsilon = 0$. Assume now $|x'_h| \geq 2^{-20}$, thus we use the else branch of the algorithm. Since $|x'_h| < 2^{14.7}$, we have $2^{32} \leq C + x'_h < 3 \cdot 2^{31} + 2^{14.7}$, thus S lies in the binade $[2^{32}, 2^{33})$, with $\text{ulp}(S) = 2^{-20}$. We thus have $S = C + x'_h + \tau$ with $|\tau| < 2^{-20}$, the sign of τ depending on the rounding mode. In line 5, we read the bits of $x'_h + \tau$ as a two's complement integer, where $|x'_h + \tau| < 2^{15}$: $S = C + n + f \cdot 2^{-20}$. By Sterbenz's lemma, the subtraction $S - C$ is exact, thus $r = \circ(x'_h - (S - C))$. Since $|x'_h| \geq 2^{-20}$, $\text{ulp}(x'_h) \geq 2^{-72}$, thus $-\tau = x'_h - (S - C)$ is an integer multiple of 2^{-72}, with $|\tau| < 2^{-20}$, thus $x'_h - (S - C)$ is exact too, and $r = -\tau = x'_h - (n + f \cdot 2^{-20})$. Since the extraction of (n, f) is exact, and likewise for the computation of r, the only rounding error might occur in the FASTTWOSUM call, which we denote by ϵ: $r + x'_\ell = y_h + y_\ell + \epsilon$. It yields $x'_h + x'_\ell = n + f \cdot 2^{-20} + y + \epsilon$.

Now let us bound $|\epsilon|$ and $|y_\ell|$. If $|r| \geq |x'_\ell|$, the precondition of FASTTWOSUM is fulfilled, and using Theorem 6 of [6], the FASTTWOSUM error is bounded by $2^{-105}\,\text{ufp}\,y_h$. Since $y_h = \circ(r + x'_\ell)$ with $|r| = |\tau| < 2^{-20}$ and $|x'_\ell| < 2^{-37.141}$, this yields $|y_h| < 2^{-19.9999}$ and $|\epsilon| \leq 2^{-125}$. If e is the rounding error in $y_h = \circ(r + x'_\ell)$, i.e., $y_h = r + x'_\ell - e$, we know (see for example the proof of Theorem 1 in [2]) that $y_\ell = \circ(e)$; since $|e| < \text{ulp}(y_h) \leq \text{ulp}(2^{-19.9999}) = 2^{-72}$, it yields $|y_\ell| \leq 2^{-72}$.

If $|r| < |x'_\ell|$, the precondition of FASTTWOSUM is not fulfilled. However, it is known (see for example Theorem 4.3 from [9]) that FASTTWOSUM behaves normally when $|r|$ and $|x'_\ell|$ lie in the same binade. Thus since $|x'_\ell| < 2^{-37.141}$, the precondition of FASTTWOSUM is not fulfilled only when $|r| < 2^{-38}$. In that case Theorem 2 of [2] yields $|\epsilon| < 3 \cdot 2^{-53}|y_h|$. It gives $|y_h| \leq \circ(2^{-38} + 2^{-37.141}) < 2^{-36.507}$, and $|\epsilon| < 3 \cdot 2^{-89.507} < 2^{-87.922}$. Then since $y_h = r + x'_\ell + \alpha$ with $|\alpha| < \text{ulp}(y_h) \leq 2^{-89}$, and $y_h + y_\ell + \epsilon = r + x'_\ell$, we get $y_\ell + \epsilon = -\alpha$, thus $|y_\ell| < |\epsilon| + |\alpha| < 2^{-87.922} + 2^{-89} < 2^{-72}$.

Since $|y_h| < 2^{-19.9999}$ and $|y_\ell| < 2^{-72}$, we have $|y| < 2^{-19.9999} + 2^{-72} < 2^{-19.999}$. ∎

Now if we use the output x' of Algorithm 8 in Algorithm 9, combining the bounds of Lemma 5 and Lemma 6, we get when $|x'_h| \geq 2^{-20}$:

$$|n + \frac{f}{2^{20}} + y - \frac{x}{\log 2}| < 2^{-87.346},$$

and when $|x'_h| < 2^{-20}$ since $\epsilon = 0$ in Lemma 6:

$$|n + \frac{f}{2^{20}} + y - \frac{x}{\log 2}| < 2^{-123.3}.$$

By picking alternate values of C, Algorithm 9 can be adapted to different scales.

c) Subroutine LOOKUP*:* Tabulating the whole 2^{20} possible values for $2^{f/2^{20}}$ would be prohibitively expensive. We split f in base 32 as $f = \sum_{i=0}^{3} f_i \cdot 2^{5i}$ with $0 \leq f_i < 32$:

$$2^{f/2^{20}} = \prod_{i=0}^{3} 2^{f_i/2^{20-5i}}.$$

For $j = 0...3$, let t_j be a table of double-double values approximating $2^{x/2^{20-5j}}$ to nearest for $x = 0...31$. We use the following algorithm:

Algorithm 10 LOOKUP

Input: : integer $f \in [0, 2^{20} - 1]$
Output: : $z = (z_h, z_\ell)$ approximating $2^{f/2^{20}}$

split $f = \sum_{i=0}^{3} f_i 2^{5i}$ bitwise
$a_i = (a_{ih}, a_{i\ell}) \leftarrow t_i[f_i]$ for $i = 0...3$
$b = (b_h, b_\ell) \leftarrow \text{DMUL}(a_0, a_1)$
$b' = (b'_h, b'_\ell) \leftarrow \text{DMUL}(a_2, a_3)$
$z \leftarrow \text{DMUL}(b, b')$

Since each double-double needs 16 bytes of memory, the total lookup table size is only $16 \times 32 \times 4$ bytes, or 2kB. Moreover the tables are easily cache-aligned. We precomputed the tables t_j using SageMath.

Lemma 7: Subroutine LOOKUP's result $z = (z_h, z_\ell)$ yields

$$\left| z - 2^{f/2^{20}} \right| \leq 2^{f/2^{20}} \cdot 2^{-100.540}.$$

Moreover $|z_h| < 2$ and $|z_\ell| < 2^{-49.271}$.

Proof: Since the total table size is 2^{20}, we can do an exhaustive check against MPFR for each rounding mode. Then, the maximal error can be shown to be between $v =$ `0x1.5ff27a94p-101` (for rounding toward zero or downward) and $v + 2^{-130}$. This exhaustive check also provides the maximum values of $|z_h|$ and $|z_\ell|$. ∎

$$
\begin{aligned}
Q &= 1 \\
&+ (\texttt{0x1.62e42fefa39efp-1} \\
&+ \texttt{0x1.abc9e3b369936p-56}) \cdot y \\
&+ \texttt{0x1.ebfbdff82c696p-3} \cdot y^2 \\
&+ \texttt{0x1.c6b08d704a1cdp-5} \cdot y^3.
\end{aligned}
$$

Fig. 1. Degree-3 polynomial approximating 2^y for $|y| < 2^{-19.9999}$.

d) Subroutine POLY*:* We implement POLY using the degree-3 polynomial of Figure 1. This polynomial was found using Sollya and perfectly accurate evaluation of it would yield an absolute error at most $2^{-89.218}$ for $|y| < 2^{-19.999}$. We evaluate it using the following scheme:

$$Q(y) \approx 1 + y \cdot q_1 + y_h^2(q_2 + y_h \cdot q_3),$$

where q_1 decomposes into $q_{1h} + q_{1\ell}$ as double-double. The terms of order 2 and above are only computed in double-precision. Using an FMA and an independent multiply, this is particularly efficient.

Algorithm 11 POLY

Input: : $y = y_h + y_\ell$, $|y|, |y_h| < 2^{-19.999}$, $|y_\ell| < 2^{-71.63}$
Output: : $t = t_h + t_\ell$ approximating 2^y

1: $w \leftarrow \circ(y_h^2)$
2: $u \leftarrow \circ(q_3 y_h + q_2)$
3: $v \leftarrow \circ(wu)$
4: $d_h, d_\ell \leftarrow \text{DMUL}(q_{1h}, q_{1\ell}, y_h, y_\ell)$
5: $f_h, f_\ell \leftarrow \text{FASTTWOSUM}(1, v)$
6: $t_h, z \leftarrow \text{FASTTWOSUM}(f_h, d_h)$
7: $t_\ell \leftarrow \circ(z + \circ(f_\ell + d_\ell))$

Lemma 8: Let $y = (y_h, y_\ell)$ be a double-double value such that $|y|, |y_h| \leq 2^{-19.999}$ and $|y_\ell| \leq 2^{-71.63}$. Then POLY's result t is such that $|t_h| < 1.01$, $|t_\ell| < 2^{-50.948}$, and

$$|t - 2^y| \leq 2^y \cdot 2^{-89.0001}.$$

Proof: Since $|y_h| < 2^{-19.999}$, we have $|w| = \circ(y_h^2) < 2^{-39.997}$, and the rounding error on w is bounded by $\text{ulp}(2^{-39.997}) = 2^{-92}$, and $|w - y^2| \leq 2^{-92} + 2|y_h y_\ell| + y_\ell^2 \leq 2^{-90.157}$.

After $u = \circ(q_3 y_h + q_2)$, since $|q_3| < 1$ and $|q_2| < 0.2403$, we have $|q_3 y_h + q_2| < 2^{-19.999} + 0.2403 < 0.241$. Thus $|u| < 0.242$ and the rounding error on u is bounded by $\text{ulp}(0.242) = 2^{-55}$. Since we neglected $q_3 y_\ell$ which is bounded in absolute value by $2^{-71.63}$, we have $|u - (q_3 y + q_2)| < 2^{-55} + 2^{-71.63} < 2^{-54.999}$.

Now $v = \circ(wu)$ approximates $q_2 y^2 + q_3 y^3$ with $|wu| \leq 2^{-39.997} \cdot 0.242$, thus $|v| < 2^{-42.043}$, and the rounding error on v is bounded by $\text{ulp}(2^{-42.043}) = 2^{-95}$:

$$
\begin{aligned}
|v &- (q_2 y^2 + q_3 y^3)| \\
&\leq 2^{-95} + |wu - y^2(q_2 + q_3 y)| \\
&\leq 2^{-95} + |w - y^2||u + y^2|u - (q_2 + q_3 y)| \\
&< 2^{-95} + 2^{-90.157} \cdot 0.242 + 2^{-39.998} \cdot 2^{-54.999} \\
&< 2^{-91.838}.
\end{aligned}
$$

After $d_h, d_\ell \leftarrow \text{DMUL}(q_{1h}, q_{1\ell}, y_h, y_\ell)$, $d_h + d_\ell$ approximates $q_1 y$. Knowing that $|y_h| < 2^{-19.999}$ and $|y_\ell| < 2^{-71.63}$, Lemma 1 gives

$$
\begin{cases}
T = \text{RU}(\text{ulp}(\text{RU}(q_{1h} 2^{-19.999})) + q_{1\ell} 2^{-19.999}) \leq 2^{-72.72}, \\
D_\ell = \text{RU}(T + q_{1h} 2^{-71.63}) \leq 2^{-71.41}, \\
D_h = \text{RU}(q_{1h} 2^{-19.999}) \leq 2^{-20.52}.
\end{cases}
$$

The total error of the DMUL call is therefore at most

$$|d_h + d_\ell - q_1 y| < 2^{-125} + 2^{-124} + 2^{-55.25} \cdot 2^{-71.63} < 2^{-123.289}.$$

After $f_h, f_\ell \leftarrow \text{FASTTWOSUM}(1, v)$, $f_h + f_\ell$ approximates $1 + q_2 y^2 + q_3 y^3$. Since $|v| < 2^{-42.043}$, the FastTwoSum condition is fulfilled, and using Theorem 6 of [6] the rounding error is bounded by $2^{-105}\,\text{ufp}(f_h) \leq 2^{-105}$. Also, since $|f_h| < 2$ it follows $|f_\ell| < \text{ulp}(1) = 2^{-52}$:

$$|f_h + f_\ell - (1 + q_2 y^2 + q_3 y^3)| < 2^{-105} + 2^{-91.838} < 2^{-91.837}.$$

After $t_h, z \leftarrow \text{FASTTWOSUM}(f_h, d_h)$, $t_h + z + f_\ell + d_\ell$ approximates $Q(y)$. Since $|f_h| \geq \circ(1 - 2^{-42.043}) \geq 1/2$ and $|d_h| < 2^{-20.5}$, the FastTwoSum condition is fulfilled. Furthermore $|f_h + d_h| < \circ(1 + 2^{-42.043}) + 2^{-20.5} < 1.01$, and by the same argument as above, the FastTwoSum rounding error is bounded by $2^{-105}\,\text{ufp}(1.01) = 2^{-105}$, and $|z| < \text{ulp}(1.01) = 2^{-52}$:

$$
\begin{aligned}
&|t_h + z + f_\ell + d_\ell - Q(y)| \\
&\quad < 2^{-105} + 2^{-91.837} + 2^{-123.289} < 2^{-91.836}.
\end{aligned}
$$

Finally after $t_\ell \leftarrow \circ(z + \circ(f_\ell + d_\ell))$, $t_h + t_\ell$ approximates $Q(y)$. Let $\sigma = \circ(f_\ell + d_\ell)$. Since $|f_\ell| < 2^{-52}$ and $|d_\ell| < 2^{-71.4}$, it follows $|\sigma| < 2^{-51.9}$ and the rounding error on σ is bounded by $\text{ulp}(2^{-51.9}) = 2^{-104}$. Since $|z| < 2^{-52}$, we have $|z + \sigma| < 2^{-52} + 2^{-51.9} < 2^{-50.949}$, thus $|t_\ell| < 2^{-50.948}$ and the rounding error on t_ℓ is bounded by $\text{ulp}(2^{-50.948}) = 2^{-103}$:

$$|t_\ell - (z + f_\ell + d_\ell)| < 2^{-104} + 2^{-103} < 2^{-102.415}.$$

Summarizing we get:

$$|t_h + t_\ell - Q(y)| < 2^{-91.836} + 2^{-102.415} < 2^{-91.835}.$$

Since $Q(y)$ approximates 2^y for $|y| < 2^{-19.999}$ with absolute error bounded by $2^{-89.218}$, and using $2^y > 0.999999$:

$$|t_h + t_\ell - 2^y| < 2^{-91.835} + 2^{-89.218} < 2^{-89.0001} \cdot 2^y.$$

■

We can now state the main theorem:

Theorem 1: Given $x \in \mathbb{F}_{64}$, $2^{-64} \le |x| < 2^{14}$, the integer n and the double-double number s computed by Algorithm 7 satisfy:

$$e^x = 2^n \cdot s \cdot (1 + \mu)$$

with $|\mu| < 2^{-87.193}$, $2^{-9.62 \cdot 10^{-7}} < s_h < 2.03$ and $|s_\ell| < 2^{-48.2}$.

Proof: From Lemma 5, the double-double number $x' = x'_h + x'_\ell$ computed by Algorithm 8 satisfies $|x'_h| < 2^{14.7}$, $|x'_\ell| < 2^{-37.141}$ and $x' = x/\log 2 + \tau$ with $|\tau| < 2^{-88.951}$. Thus Lemma 6 applies and (n, f, y) satisfy $x' = n + f/2^{20} + y + \epsilon$ with $0 < f < 2^{20}$, $|y|, |y_h| < 2^{-19.999}$, $|y_\ell| \le 2^{-71.63}$ and $|\epsilon| < 2^{-87.922}$. It follows:

$$e^x = 2^n \cdot 2^{f/2^{20}} \cdot 2^y \cdot 2^{\epsilon - \tau}.$$

Now from Lemma 7, the result z from subroutine LOOKUP satisfies $z = 2^{f/2^{20}}(1 + \eta)$ with $|\eta| < 2^{-100.540}$, and by Lemma 8, the double-double value t computed by subroutine POLY satisfies $t = 2^y(1 + \alpha)$ with $|\alpha| < 2^{-89.0001}$. thus

$$e^x = 2^n \cdot z \cdot t \cdot \frac{2^{\epsilon - \tau}}{(1 + \eta)(1 + \alpha)}.$$

Using Lemma 1 on the DMUL(t, z) call using the postconditions of Lemma 7 and 8 and renaming A and B the quantities from the lemma to avoid conflicts, we have

$$\begin{cases} A = \mathrm{RU}(2.02 \cdot 2^{-52} + 2 \cdot 2^{-50.948}) \le 2^{-49.3} \\ B \le \mathrm{RU}(A + 1.01 \cdot 2^{-49.271}) \le 2^{-48.2}, \end{cases}$$

so that the last DMUL's error is bounded by $2^{-102} + 2^{-101} + 2^{-50.948} \cdot 2^{-49.271} < 2^{-99.313}$.

Since $|z| \ge 1$ (by looking at the table values), and $|t| \ge 2^{-2^{-19.999}}(1 - 2^{-89.0001}) \ge 2^{-9.6 \cdot 10^{-7}}$ we deduce the relative error of DMUL is bounded by $2^{-99.313}/2^{-9.6 \cdot 10^{-7}} < 2^{-99.312}$:

$$s = tz(1 + \beta) \quad \text{with } |\beta| < 2^{-99.312}.$$

Note that in this way, $s \ge 2^{-9.6 \cdot 10^{-7}}(1 - 2^{-99.312}) \ge 2^{-9.61 \cdot 10^{-7}}$

The RECONSTRUCT procedure consists of a FASTTWOSUM on the two components of s followed by Algorithms 5 and 6. Lastly, the exponent n is added. We will call r_0 the result of this procedure before taking n into account, and r the return value. Splitting s as s_h, s_ℓ, the previous DMUL ensures that $|s_\ell| \le 2^{-48.2}$. Since $|s_h| \ge 2^{-9.61 \cdot 10^{-7}} - 2^{-48.2} \ge 2^{-48.2}$, the error of the FASTTWOSUM is at most 2^{-105} of the result. Applying the error bound from Algorithm 5, we get that the return value r_0 follows

$$r_0 = s(1 + \gamma) \quad \text{with } |\gamma| < (1 + 2^{-105})(1 + 2^{-126}) - 1.$$

We have $|\gamma| \le 2^{-104.999}$.

Summarizing, we get:

$$e^x = r \cdot \frac{2^{\epsilon - \tau}}{(1 + \eta)(1 + \alpha)(1 + \beta)(1 + \gamma)},$$

which implies $e^x = r \cdot (1 + \mu)$ with $|\mu| < 2^{-87.193}$. ∎

Procedure RECONSTRUCT leverages Lemma 4 to determine whether the result is correctly rounded. Having a from Algorithm 6, if $|a| > 2^{41}$ then the distance from nearest rounding boundary is at least $a \cdot 2^{-64}\,\mathrm{ulp}_{64}(r_0) \ge 2^{-87} r_0$ which ensures r_0 then r is correctly rounded. Otherwise, the fast path is unconclusive.

B. Accurate path

The accurate path follows the same computation scheme as the fast path, leveraging the extra software precision to attain a relative error bound of $2^{-167.006}$.

To reduce total lookup table size we reuse the fast path's lookup tables using the "accurate tables" trick from [7]. Let $1 \le k \le 4$ and $0 \le t < 32$. Assume the fast path's lookup tables approximated $2^{t/2^{5k}}$ as x. We then know that

$$\left| x - 2^{t/2^{5k}} \right| \le 2^{-107}.$$

As a consequence, there must be some small s such that

$$x = 2^s \cdot 2^{t/2^{5k}}.$$

By storing a 62-bit approximation of s, we can make the lookup phase output an approximation pair (s', y) with $y \approx 2^{f/2^{20} + s'}$ and s' small. By adding s' to the reduced value, this allows us to get a lookup accuracy of $107 + 62 = 169$ bits using only 4 tables of 32 words of 64 bits, which fit in 1kB.

C. Computing hard-to-round cases with BaCSeL

We have computed hard-to-round inputs using BaCSeL. BaCSeL [3] is a software tool to compute hard-to-round cases of a mathematical function. BaCSeL takes as input an integer m, and outputs all n-bit floating-point numbers x in $[x_0, x_1)$ such that:

$$|f(x) - \circ(f(x))| < 2^{-m}\,\mathrm{ulp}(f(x)),$$

where $\circ(\cdot)$ is any rounding mode. The complexity of the underlying algorithm depends on several factors, but the larger is m, the faster is the algorithm.

a) Normal output: Let x_2 be the smallest long double value such that $\exp x > 2^{16384}$ and x_1 the largest such that $\exp x_1 < 2^{-16382}$. Notice that for $|x| < 2^{-65}$, $\exp x$ rounds to the same value as $1 + x$, thus we only have to search for hard-to-round inputs in the ranges $[x_1, -2^{-65}]$ and $[2^{-65}, x_2)$. For $|x|$ large, the efficiency of the SLZ algorithm used by BaCSeL decreases. For this reason, we further subdivided the search into sub-ranges:

- for $-2^4 < x \le -2^{-65}$ and $2^{-65} \le x <$ `0x1.484p+9`, we search worst cases with at least 54 identical bits after the round bit. We found 158,662 such worst cases. The largest number of identical bits after the rounding bit is 126 for $x =$ `0x1.fffffffffffffep-64`. Apart from such inputs very close to zero and with a special form due to the Taylor expansion of exp, the largest number of identical bits after the rounding bit is 75 for $x = -$`0x1.625ac7bfa54aba72p-14`;
- for $x_1 \le x \le -2^4$ and `0x1.484p+9` $\le x < x_2$, we search worst cases with at least 101 identical bits after the round bit. We found no such worst case.

b) Subnormal output: Let x_0 be the smallest long double such that $\exp x_0 \geq 2^{-16445}$. For $x_0 \leq x \leq x_1$, $\exp x$ lies in the subnormal range $[2^{-16445}, 2^{-16382})$. For each exponent e, $-16444 \leq e \leq -16382$, we compute the largest interval (ℓ, h) such that $2^{e-1} \leq \exp(\ell) \leq \exp(h) < 2^e$, and search with BaCSeL inputs in this interval with at least 101 identical bits after the round bit (which has weight 2^{-16446} for all intervals). We found no such input.

c) Correctness of the accurate path:

Lemma 9: Assume a floating-point format with a precision of p bits, and an accurate path with relative error less than 2^k. Then if x is such that $f(x)$ is at distance at least 2^{-m} ulp from a rounding boundary, then rounding the approximation y of the accurate path delivers the correct rounding of $f(x)$ as long as:

$$m \leq -k - p - 1 + \log_2(1 - 2^k). \qquad (1)$$

Moreover this remains true in the subnormal range.

Proof: Let x as in the statement of the lemma, and z a $(p+1)$-bit floating-point number. We then have:

$$|f(x) - z| \geq 2^{-m} \operatorname{ulp}_p(z) > 2^{-m-p}|z|.$$

The approximation y of $f(x)$ computed by the accurate path satisfies $|f(x) - y| < \varepsilon$ with $\varepsilon = 2^k|y|$. Assume the confidence interval $[y - \varepsilon, y + \varepsilon]$ crosses a rounding boundary z. Then:

$$|z - y| \leq 2^k|y|, |f(x) - y| < 2^k|y|, |f(x) - z| > 2^{-m-p}|z|.$$

By the triangular inequality, it follows $2^{k+1}|y| > 2^{-m-p}|z| \geq 2^{-m-p}(1 - 2^k)|y|$. This contradicts the hypothesis on m, thus the confidence interval $[y - \varepsilon, y + \varepsilon]$ cannot contain any rounding boundary, which proves that any number in this interval rounds to the same value. Therefore rounding y yields the correct rounding of $f(x)$.

In the subnormal range, the output "local" precision p' satisfies $p' \leq p$: if Eq. (1) holds for p, it also holds for p'. ∎

In the `expl` case, the relative error of the accurate path is less than 2^k with $k = -167.006$, we have $p = 64$, thus from Lemma 9 we find that $m \leq 102$ suffices. This corresponds to `-m 102` in BaCSeL syntax, which means searching all inputs such that $|\exp x - z| < 2^{-102} \operatorname{ulp}_{64}(z)$, where z is a 65-bit floating-point number (exact double extended or midpoint). The correctness of the accurate path follows from the following remarks:

- if $f(x)$ has at least 101 identical bits after the round bit, x will be found by our search with BaCSel. It thus suffices to check that $f(x)$ is correctly rounded by the accurate path, and otherwise treat x as exceptional case;
- otherwise, $f(x)$ has less than 101 identical bits after the round bit, and by Lemma 9 it is correctly rounded.

V. Results and Conclusion

Using the previously-described methods, we implemented correctly-rounded versions of `expl`, `powl` and `log2l` in CORE-MATH [10]. We expect our methods to be applicable to a wide array of standard functions.

	`expl`	`powl`	`log2l`
CORE-MATH	47.5	165.7	44.9
Intel Math Library (2025.0.0)	64.2	288.4	83.1
GNU Libc 2.40	127.1	761.6	65.0
Openlibm 0.8.5	151.5	640.1	151.1
Musl 1.2.5	115.0	546.5	47.3

Fig. 2. Reciprocal throughput (in cycles) of some double extended precision functions on a Intel Xeon Silver 4214 with GCC 14.2.0.

We certified the correctness of our implementations using BaCSeL, and tested their performance with the CORE-MATH benchmark suite. We consistently obtained better performance than the other mathematical librairies with double extended support (see Figure 2). We attribute this to our use of double-double arithmetic in the fast path, which has a very high success rate (on 10^8 random inputs in $[-10, 10]$, we get 19 failures of the rounding test, thus a probability of about 2^{-22}). Other implementations extensively use x87 features, which in modern processors are badly pipelined and slow.

The GNU Libc and the Intel Math Library use complex x87 features such as `FSQRT` which are not completely specified. Our code only uses FMA operations, which are completely specified. Because the correctness of our algorithms is based on the double precision FMA, we expect them to be portable across architectures, operating systems and processor manufacturers.

Acknowledgements. The authors thank the first two anonymous referees for their very useful comments.

References

[1] Brisebarre, N., Hanrot, G., Muller, J.-M., and Zimmermann, P. Correctly-rounded evaluation of a function: why, how, and at what cost? https://hal.science/hal-04474530, 2025.

[2] Corbineau, S., and Zimmermann, P. Note on FastTwoSum with Directed Roundings. https://inria.hal.science/hal-03798376, 2024.

[3] Hanrot, G., Lefèvre, V., Stehlé, D., and Zimmermann, P. The BaCSeL software tool. https://gitlab.inria.fr/zimmerma/bacsel.

[4] Hubrecht, T., Jeannerod, C.-P., and Zimmermann, P. Towards a correctly-rounded and fast power function in binary64 arithmetic. In *Proceedings of Arith'2023* (2023), pp. 111–118.

[5] Jeannerod, C.-P., Muller, J.-M., and Zimmerman, P. On various ways to split a floating-point number. In *Proceedings of Arith'2018* (2018), pp. 53–60.

[6] Jeannerod, C.-P., and Zimmermann, P. FastTwoSum revisited. In *Proceedings of Arith'2025* (2025). Preliminary version at https://inria.hal.science/hal-04875749.

[7] Markstein, P. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard professional books. Prentice Hall PTR, 2000.

[8] Muller, J.-M. On the definition of ulp(x). Research Report RR-5504, LIP RR-2005-09, INRIA, LIP, Feb. 2005.

[9] Muller, J.-M., Brunie, N., de Dinechin, F., Jeannerod, C.-P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N., and Torres, S. *Handbook of Floating-point Arithmetic (2nd edition)*. Birkhäuser Basel, 2018.

[10] Sibidanov, A., Zimmermann, P., and Glondu, S. The CORE-MATH Project. In *Proceedings of Arith'2022* (Sept. 2022), pp. 26–34.

[11] Ziv, A. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Trans. Math. Softw. 17*, 3 (1991), 410–423.