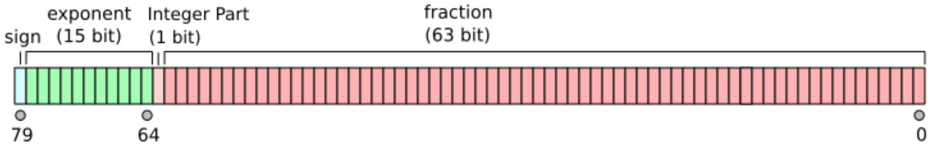
Correct rounding in Double Extended Precision

Sélène Corbineau and Paul Zimmermann

ARITH 2025, May 4-7, 2025

Plan

Double Extended Precision

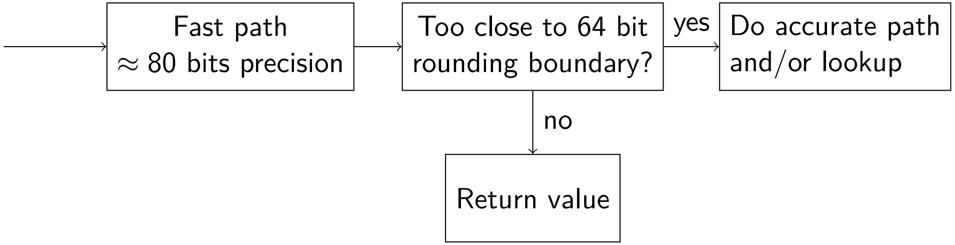


No correctly rounded routines available in double extended precision.
Existing routines are slow.

Figure by BillF4, CC BY-SA 3.0

Strategy

We follow Ziv's method to achieve correct rounding:


Fast path
≈ 80 bits precision
Too close to 64 bit rounding boundary?
yes
Do accurate path and/or lookup
no
Return value

We implement the fast path with double precision arithmetic, avoiding x87 instructions.

Plan

Current state-of-the-art

GNU libc, development version, with patch to benchmark some long double functions, on Intel Core i7-8700 with gcc 14.2.0. Timings in cycles.

function	min	mean	max
fmal	556	678	1030
expl	137	142	351
log2l	57	174	554
powl	700	743	1156

Plan

Double-double arithmetic

Double-double arithmetic uses pairs $(a, b) \in \mathbb{F}_{64}^2$ to represent $a + b$.

- we have product, sum primitives with explicit error bounds
- when $|b| < \mathrm{ulp}(a)$ table lookups are easy

We can expect $\approx 53 \times 2 = 106$ bits of precision, more than `long double`'s 64.

Converting from double extended precision

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

This only works if the exponent of x fits in the `double` exponent range.
Implemented routines deal with large/small inputs differently:

- exponentials saturate to $0, +\infty$ or 1
- logarithms keep track of x's exponent separately

Converting to double extended precision

Starting from $(a, b) \in \mathbb{F}_{64}^2$ with $|b| < \mathrm{ulp}(a)$:

1. Compute $q \approx a + b$ on 128 bits
2. Round q to `long double` and compute distance to rounding boundary

Computing q

Input: $a, b \in \mathbb{F}_{64}$ not denormals with $|b| < \mathrm{ulp}(a)$
Output: $q = (q_s, q_e, q_m) \in \mathbb{F}_{128}$ such that $(-1)^{a_s} 2^{q_e - 127} \cdot q_m \approx a + b$

```
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:     $q_e \leftarrow q_e - 1$
5:     $q_m \leftarrow 2 q_m$
   return $q = (a_s, q_e, q_m)$
```

Assumption $|b| < \mathrm{ulp}(a)$ ensures no overflow in line 2.
There is a small error due to truncation.

Rounding q

Input: $q = (q_s, q_e, q_m)$
Output: $(y, \delta) \in \mathbb{F}_{64} \times \mathbb{Z}_{64}$ where y rounds q upwards and δ is a scaled rounding error.

```
 1: write $q_m = m_h 2^{64} + m_\ell$ with $0 \le m_h, m_\ell < 2^{64}$
 2: $\delta \leftarrow m_\ell \operatorname{cmod} 2^{64}$                    ▷ $-2^{63} \le \delta < 2^{63}$
 3: **if** $m_\ell \neq 0$ and $q_s > 0$ **then**
 4:     $m_h \leftarrow m_h + 1$
 5:     **if** $m_h = 2^{64}$ **then**
 6:         $q_e \leftarrow q_e + 1, \quad m_h \leftarrow 2^{63}$
 7:         $\delta \leftarrow \delta/2$                             ▷ rounded towards zero
 8: **if** $q_e \ge 16384$ **then**
 9:     **return** $((-1)^{q_s}\infty, \delta)$
10: **return** $(\mathbb{F}_{64}(q_s, q_e, m_h), \delta)$
```

Plan

Evaluation strategy

Inputs with exponent < -64 or ≥ 14 round trivially.
Else, let $x' = x/\log 2$ and use

$$x' = n + \frac{f}{2^{20}} + (y_h + y_\ell)$$
$$e^x = 2^{x'} = 2^n \cdot 2^{f/2^{20}} \cdot 2^{y_h + y_\ell}$$

Splitting x'

$$x' = n + \frac{f}{2^{20}} + (y_h + y_\ell)$$

1. Split x as double-double
2. Multiply by $1/\log 2$ as double-double
3. Clobber the high bits to get n, f
4. Normalize the remainder as y_h, y_ℓ with $|y_\ell| < \mathrm{ulp}(y_h)$

Evaluating the exponential

$$e^x = 2^{x'} = 2^n \cdot 2^{f/2^{20}} \cdot 2^{y_h + y_\ell}$$

1. $2^{f/2^{20}}$ is computed with a few table lookups
2. $2^{y_h + y_\ell}$ is evaluated by a degree 3 polynomial

All computations are done in double-double. We add n to the final exponent, after rounding (subnormals are treated apart).

Reconstruction

We round the previous double-double value to extended precision.
We compare $|\delta|$ to the relative computation error:

- If $|\delta| > 2^{41}$ we can guarantee correct rounding
- Otherwise, we go to the accurate path.

In practice, the test fails with probability $\approx 2^{-22}$.

Computing hard-to-round cases

We use the BaCSeL software tool (`https://gitlab.inria.fr/zimmerma/bacsel`).

- for $-2^4 < x < -2^{-65}$ and $2^{-65} \leq x <$ `0x1.484p+9`, we found 158,662 inputs with at least 54 identical bits after the round bit. Apart special cases, the largest number is 75 for $x =$ `-0x1.625ac7bfa54aba72p-14`.
- for $x \leq -2^4$ and `0x1.484p+9` $\leq x$, we search hard-to-round cases with at least 101 identical bits after the round bit. We found none.

Accurate path

Same scheme as the fast path.

Home-made 192-bit arithmetic (using 3 integer words of 64 bits).

Relative error bound is $2^{-167.006}$.

Reuse the fast path lookup tables using Markstein's “accurate table” trick.

Use the 7th degree Taylor polynomial.

Plan

Double extended precision and correct rounding

Current state of the art

Converting between double extended and double-double

An example: `expl`

Results

Implementation

We implemented using this scheme in CORE-MATH:

- `expl`, `exp2l`
- `log2l`
- `powl`, which was more challenging due to the dynamic ranges involved.

Other functions (`cbrtl`, `hypotl`, `rsqrtl`) were implemented using a different scheme.

Performance

	`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

Figure: Reciprocal throughput in cycles on an Intel Xeon Silver 4214 and GCC 14.2.0

Our routines only use a few kilobytes of lookup tables.

Conclusion

- We implemented correctly rounded routines for double extended precision
- Avoiding x87 enables fast and more portable double extended precision routines
- Might be used even on processor without double extended support
- Still many functions to implement...