Optimized Binary64 and Binary128 Arithmetic with GNU MPFR
(common work with Vincent Lefèvre)

Paul Zimmermann
Introducing the GNU MPFR library

- a software implementation of \textit{binary} IEEE-754 (decimal implementation provided by decNumber from Mike Cowlishaw)

- variable/arbitrary precision (up to the limits of your computer)

- each variable has its own precision: mpfr_init (a, 35)

- global user-defined exponent range (might be huge): mpfr_set_emin (-123456789)

- mixed-precision operations: $a \leftarrow b - c$ where $a$ has 35 bits, $b$ has 42 bits, $c$ has 17 bits

- correctly rounded mathematical functions (\textit{exp}, \textit{log}, \textit{sin}, \textit{cos}, ...) as in Section 9 of IEEE 754-2008
History

- 2000: first public version;
- 2008: MPFR is used by GCC 4.3.0 for constant folding:
  \[
  \text{double } x = \sin (3.14);
  \]
- 2009: MPFR becomes GNU MPFR;
- 2016: 4th developer meeting in Toulouse.
- mpfr.org/pub.html mentions 2 books, 27 PhD theses, 59 papers citing MPFR
SageMath version 7.6, Release Date: 2017-03-25
Type `'notebook()'` for the browser-based notebook interface.
Type `'help()'` for help.

sage: x=1/7; a=10^-8; b=2^24
sage: RealIntervalField(24)(x+a*sin(b*x))
[0.142857119 .. 0.142857150]
Now in english!
This work

- concentrates on small precision (1 to 2 machine-words)
- all operands have same precision
- basic operations: add, sub, mul, div, sqrt
- get the fastest possible software implementation
- while keeping the same user interface
Correct Rounding

Definition: we compute the floating-point value closest to the exact result, with the given precision and rounding modes (following IEEE-754).

RNDN: to nearest (ties to even);

RNDZ: toward zero, RNDA: away from zero;

RNDD: toward $-\infty$, RNDU: toward $+\infty$.

Only one possible conforming result: the correct rounding.
Notations

MPFR uses GMP’s mpn layer for the internal representation of significands.

*limb*: a GMP word (in general 32 or 64 bits)

We will assume here a *limb* has 64 bits.

In a 64-bit limb, we call “bit 1” the most significant bit, and “bit 64” the least significant one.
Representation of MPFR numbers (*mpfr_t*)

- precision $p \geq 1$ (in bits);
- sign ($-1$ or $+1$);
- exponent (between $E_{\text{min}}$ and $E_{\text{max}}$), also used to represent special numbers ($\text{NaN}$, $\pm\infty$, $\pm 0$);
- significand (array of $\lceil p/64 \rceil$ limbs), defined only for regular numbers (neither $\text{NaN}$, nor $\pm\infty$ and $\pm 0$, which are singular values).

The most significant bits are shown on the left.
Regular numbers are *normalized*: the most significant bit of the most significant limb should be 1.
Example, $x = 17$ with a precision of 10 bits and limbs of 6 bits is represented as follows:

```
precision +1 5 100010 000000
```

```
+1
sign

5
exponent

100010
limb 1

000000
limb 0
```
**Round bit and sticky bit**

\[ v = \underbrace{xxx...yyy}_{m \text{ of } p \text{ bits}} \underbrace{r}_{\text{round bit}} \underbrace{sss...}_{\text{sticky bit}} \]

The *round bit* \( r \) is the value of bit \( p + 1 \) (where bit \( p \) is the least significant bit of the significand).

The *sticky bit* \( s \) is zero iff \( sss... \) is zero.

The *round bit* and *sticky bit* enable us to determine correct rounding for all rounding modes:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>toward zero</th>
<th>to nearest ( m )</th>
<th>away from zero ( m )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>( m )</td>
<td>( m )</td>
<td>( m )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>( m )</td>
<td>( m )</td>
<td>( m + 1 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>( m )</td>
<td>( m + (m \mod 2) )</td>
<td>( m + 1 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( m )</td>
<td>( m + 1 )</td>
<td>( m + 1 )</td>
</tr>
</tbody>
</table>
The function mpfr_add

The function mpfr_add\((a, b, c)\) works as follows \((a \leftarrow b + c)\):

- first check for singular values \((NaN, \pm Inf, \pm 0)\);
- if \(b\) and \(c\) have different signs, call the subtraction code;
- if \(a, b, c\) have same precision, call mpfr_add1sp;
The (new) function mpfr_add1sp

- if $p < 64$, call mpfr_add1sp1;
- if $p = 64$, call mpfr_add1sp1n;
- if $64 < p < 128$, call mpfr_add1sp2;
- otherwise execute the generic code for operands of same precision.

Note: $p = 128$ uses the generic code, prefer $p = 127$ if possible.
The function mpfr_add1sp1

Case 1, $e_b = e_c$:

\[
\begin{align*}
    b &= \boxed{110100} \\
    c &= \boxed{111000}
\end{align*}
\]

\[
a0 = (bp[0] >> 1) + (cp[0] >> 1);
\]

\[
bx ++;
\]

\[
rb = a0 \& (MPFR_LIMB_ONE << (sh - 1));
\]

\[
ap[0] = a0 \oplus rb;
\]

\[
sb = 0;
\]

Since $b$ and $c$ are normalized, the most significant bits from $bp[0]$ and $cp[0]$ are 1.

Thus the addition of $bp[0]$ and $cp[0]$ always produces a carry, and the exponent of $a$ is $e_b + 1$ (here $bx + 1$).
\[ b = \underline{110100} \]

\[ c = \underline{111000} \]

\[
a_0 = (bp[0] >> 1) + (cp[0] >> 1);
bx ++;
rb = a_0 \& (\text{MPFR\_LIMB\_ONE} \ll (sh - 1));
ap[0] = a_0 \ xor \ rb;
sb = 0;
\]

The sum might have \( p + 1 \) significant bits, but since \( p < 64 \) (\( p < 6 \) in the example), it always fits into 64 bits.

\( sh \) is the number \( 64 - p \) of unused bits, here \( 6 - p = 2 \).

The \textit{round bit} is bit \( p + 1 \) from the sum, the \textit{sticky bit} is always zero.

We might have an \textit{overflow}, but no \textit{underflow}.
The function mpfr_sub

The function mpfr_sub\((a, b, c)\) works as follows \((a \leftarrow b - c)\):

- first check for singular values \((NaN, \pm Inf, \pm 0)\);
- if \(b\) and \(c\) have different signs, call the addition code;
- if \(b\) and \(c\) have same precision, call mpfr_sub1sp;
- otherwise call the generic code mpfr_sub1.
The function mpfr_sub1sp

- if $p < 64$, call mpfr_sub1sp1;
- if $p = 64$, call mpfr_sub1sp1n;
- if $64 < p < 128$, call mpfr_sub1sp2;
- otherwise execute the generic code for the subtraction of operands of same precision.

Note: as for addition, prefer $p = 127$ to $p = 128$ if possible.
The function mpfr\_sub1sp1

- if exponents differ, swap $b$ and $c$ if necessary, so that $e_b \geq e_c$;
- case 1: $e_b = e_c$;
- case 2: $e_b > e_c$. 
Case 1, $e_b = e_c$:

$$b = \boxed{110100}$$

$$c = \boxed{111000}$$

compute $bp[0] - cp[0]$ and store the result in $ap[0]$, which then equals $bp[0] - cp[0] \mod 2^{64}$;

if $ap[0] = 0$, the result is 0;

if $ap[0] > bp[0]$, a borrow occurred, we thus have $|c| > |b|$: change $ap[0]$ into $-ap[0]$ and change the sign of $a$;

otherwise no borrow occurred, thus $|c| < |b|$;

compute the number $k$ of leading zeros of $ap[0]$, shift $ap[0]$ by $k$ bits to the left and decrease the exponent by $k$;

in this case the round bit and sticky bit are always 0.

We might have an underflow, but no overflow: $|a| \leq \max(|b|, |c|)$. 

The function mpfr_mul(a,b,c)

\[ a \leftarrow o(b \cdot c) \]

- if \( p_a = p_b = p_c < 64 \), call mpfr_mul_1;
- if \( p_a = p_b = p_c = 64 \), call mpfr_mul_1n;
- if \( 64 < p_a = p_b = p_c < 128 \), call mpfr_mul_2;
- otherwise use the generic code.
The function mpfr\_mul\_1

\[
a ← o(b \cdot c)
\]

\(a, b, c: \) at most one limb (minus 1 bit):

\[
h \cdot 2^{64} + ℓ ← bp[0] \cdot cp[0] \quad (\text{umul\_ppmm})
\]

Since \(2^{63} ≤ bp[0], cp[0] < 2^{64}\), we have \(2^{62} ≤ h\).

If \(h < 2^{63}\), shift \(h, ℓ\) of one bit to the left, and decrease the exponent.

The round bit is bit \(p + 1\) of \(h\) \((p < 64)\).

The sticky bit is formed by the remaining bits from \(h\) (none if \(p = 63\)) and those of \(ℓ\).

Both underflow and overflow might happen.

Beware: MPFR considers underflow after rounding (with an infinite exponent range).
Underflow before vs after rounding

Assume $bc = 0.\overbrace{111...111}^{p \text{ bits}} 101 \cdot 2^{E_{\min} - 1}$.

With underflow before rounding, there is an underflow since the exponent of $bc$ is $E_{\min} - 1$.

With underflow after rounding, and rounding to nearest, $\circ(bc) = 0.100...000 \cdot 2^{E_{\min}}$, and there is no underflow since the exponent of $\circ(bc)$ is $E_{\min}$.
The function mpfr_div(a,b,c)

\[ a \leftarrow \circ(b/c) \]

- if \( p_a = p_b = p_c < 64 \), call mpfr_div_1;
- if \( p_a = p_b = p_c = 64 \), call mpfr_div_1n;
- if \( 64 < p_a = p_b = p_c < 128 \), call mpfr_div_2;
- otherwise use the generic code.
The function mpfr\_div\_1

\[ a \leftarrow \circ(b/c) \]

We have \( p_a = p_b = p_c < 64 \):

1. \( bp[0] \geq cp[0] \): one extra quotient bit;
2. \( bp[0] < cp[0] \): no extra quotient bit.
Algorithm DivApprox1

Input: integers $u, v$ with $0 \leq u < v$ and $\beta/2 \leq v < \beta$.
Output: integer $q$ approximating $u\beta/v$.

1: compute an approximate inverse $i$ of $v$, verifying

$$i \leq \lfloor (\beta^2 - 1)/v \rfloor - \beta \leq i + 1$$

2: $q = \lfloor iu/\beta \rfloor + u$

Note: here we have $\beta = 2^{64}$.

The computation of the approximate inverse is done by a variant of the GMP macro invert_limb (Möller and Granlund, Improved division by invariant integers, IEEE TC, 2011).
**Theorem**

The approximate quotient computed by Algorithm DivApprox1 satisfies

\[ q \leq \lfloor \frac{u\beta}{v} \rfloor \leq q + 2. \]

Consequence: we can determine the correct rounding of \( u/v \), except if the last \( s_h-1 \) bits from \( q \) are 000..000, 111..111 or 111..110.

In this (rare) case, to improve the worst case latency, we start from the approximation \( q \).
The function mpfr_sqrt(r,u)

\[ r \leftarrow \circ(\sqrt{u}) \]

- if \( p_r = p_u < 64 \), call mpfr_sqrt1;
- if \( p_r = p_u = 64 \), call mpfr_sqrt1n;
- if \( 64 < p_r = p_u < 128 \), call mpfr_sqrt2;
- otherwise use the generic code.
Algorithm RecSqrtApprox1

**Input:** integer $d$ with $2^{62} \leq d < 2^{64}$.

**Output:** integer $v_3$ approximating $s = \lfloor 2^{96} / \sqrt{d} \rfloor$.

1. $d_{10} = \lfloor 2^{-54} d \rfloor + 1$
2. $v_0 = \lfloor \sqrt{2^{30} / d_{10}} \rfloor$  
   (table lookup)
3. $d_{37} = \lfloor 2^{-27} d \rfloor + 1$
4. $e_0 = 2^{57} - v_0^2 d_{37}$
5. $v_1 = 2^{11} v_0 + \lfloor 2^{-47} v_0 e_0 \rfloor$
6. $e_1 = 2^{79} - v_1^2 d_{37}$
7. $v_2 = 2^{10} v_1 + \lfloor 2^{-70} v_1 e_1 \rfloor$
8. $e_2 = 2^{126} - v_2^2 d$
9. $v_3 = 2^{33} v_2 + \lfloor 2^{-94} v_2 e_2 \rfloor$

Remark: if a *table lookup* is faster than a multiplication, we might tabulate $v_0^2$ at step 4.
Theorem

The value $v_3$ returned by RecSqrtApprox1 differs by at most 8 from the inverse square root:

$$v_3 \leq s := \lfloor 2^{96}/\sqrt{d} \rfloor \leq v_3 + 8.$$
Algorithm SqrtApprox1

**Input:** integer $n$ with $2^{62} \leq n < 2^{64}$.

**Output:** integer $r_0$ approximating $\sqrt{2^{64}} n$.

1. compute an integer $x$ approximating $2^{63}/\sqrt{n}$ with

   $$x \leq 2^{63}/\sqrt{n}$$

2. $y = \lfloor \sqrt{n} \rfloor$ (reusing the approximation $x$)

3. $z = n - y^2$

4. $t = \lfloor 2^{-32}xz \rfloor$

5. $r_0 = y \cdot 2^{32} + t$

**Theorem**

*If the approximation $x$ at step 1 is the value $v_2$ of Algorithm RecSqrtApprox1, then Algorithm SqrtApprox1 returns $r_0$ such that*

$$r_0 \leq \lfloor \sqrt{2^{64}} n \rfloor \leq r_0 + 7.$$
**The function mpfr_sqrt1**

Input: \(2^{63} \leq u < 2^{64}\) representing a number of \(p < 64\) bits (most significant bit set to 1).

- if the associated exponent is odd, shift \(u\) by one bit to the right;
- now \(2^{62} \leq u < 2^{64}\). Call `__gmpfr_sqrt_limb_approx`, which implements SqrtApprox1, and computes \(r_0\) such that
  \[
  r_0 \leq \lfloor \sqrt{2^{64}u} \rfloor \leq r_0 + 7;
  \]
- if the \(\text{sh}-1\) least significant bits of \(r_0\) are not 000..000, 111..111 (-1), 111..110 (-2), ..., 111..011 (-5), 111..010 (-6), 111..001 (-7), then we can determine the correct rounding;
- otherwise we compute \(r = r_0 + i\) with \(0 \leq i \leq 7\) such that
  \[
  r = \lfloor \sqrt{2^{64}u} \rfloor
  \]
  which is equivalent to:
  \[
  0 \leq 2^{64}u - r^2 \leq 2r.\]
MPFR 3.1.5 compared to MPFR 4.0-dev

aráignee.loria.fr, Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz, with GMP 6.1.2 and GCC 6.3.0.
GMP and MPFR are configured with `-disable-shared`

<table>
<thead>
<tr>
<th>MPFR 3.1.5</th>
<th>MPFR 4.0-dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits</td>
<td>53 113</td>
</tr>
<tr>
<td>mpfr_add</td>
<td>52 53</td>
</tr>
<tr>
<td>mpfr_sub</td>
<td>49 52</td>
</tr>
<tr>
<td>mpfr_mul</td>
<td>49 63</td>
</tr>
<tr>
<td>mpfr_sqr</td>
<td>74 79</td>
</tr>
<tr>
<td>mpfr_div</td>
<td>134 146</td>
</tr>
<tr>
<td>mpfr_sqrt</td>
<td>171 268</td>
</tr>
</tbody>
</table>

Timings are in cycles.
Conclusion

Speedup by a factor 2 or more until 127 bits for $\div$, $\sqrt{}$, until 191 bits for $+$, $-$, $\times$.

Will be available in MPFR 4, already available in the development version!

New algorithms for division and square root, with small and tight error bounds.

Also in paper (and MPFR 4): new RNDF rounding mode (faithful)

Detailed and public code and proofs, ready for a formal proof. Any volunteers to find a bug?