Multiple-Precision Arithmetic: from MP to MPFR

Paul Zimmermann
From MP ...
The MP package


November 1973: first working version (731101)
Version 770217: matches the TOMS publication.
1978: Augment interface added
1979: storage allocation improved, rounding options implemented, dependance on Fortran REAL eliminated, added packed numbers
Main MP Features

base \( b, t \) digits, with \( b \geq 2, t \geq 2, 8b^2 - 1 \) representable as a single-precision integer

<table>
<thead>
<tr>
<th>wordlength</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 bits</td>
<td>( 2^{22} = 4194304 ) or ( 10^6 )</td>
</tr>
<tr>
<td>36 bits</td>
<td>( 2^{16} = 65536 ) or ( 10^4 )</td>
</tr>
<tr>
<td>32 bits</td>
<td>( 2^{14} = 16384 ) or ( 10^4 )</td>
</tr>
<tr>
<td>24 bits</td>
<td>( 2^{10} ) or 1000</td>
</tr>
<tr>
<td>18 bits</td>
<td>( 2^7 ) or 100</td>
</tr>
<tr>
<td>16 bits</td>
<td>( 2^6 ) or 10</td>
</tr>
<tr>
<td>12 bits</td>
<td>( 2^4 ) or 10</td>
</tr>
</tbody>
</table>

Assumption \( 8b^2 - 1 \) representable as a single-precision integer: storage waste of about 50%

Exponent in \([−m, m]\), with \( 4m \) representable as a single-precision integer
Internal representation

The precision \( t \) in words is global for a given MP session, thus does not need to be represented (idem for base \( b \)).

Example: representation of 17 with \( b = 2^{10} \) and \( t = 2 \):

\[
\begin{array}{cccc}
\text{+1} & 0 & 17 & 0 \\
\text{word 1} & \text{word 2} & \text{word 3} & \text{word 4}
\end{array}
\]
The subroutines are machine independent and the precision is arbitrary, subject to storage limitations.

We have attempted to make it efficient at a high level by implementing good algorithms.

Compressed/packed numbers to avoid the storage waste of about 50% (at the expense of increased timings by about 1.5).
Rounding in MP

\[ \text{RNDRL} = 0 \text{ means truncated (chopped) arithmetic} \]
\[ \text{RNDRL} = 1 \text{ means rounded (to nearest) arithmetic} \]
\[ \text{RNDRL} = 2 \text{ means round down (towards } -\infty \text{)} \]
\[ \text{RNDRL} = 3 \text{ means round up (towards } +\infty \text{)} \]
Underflow and Overflow in MP

underflow: set to zero

overflow: fatal error

no Infinity, no Not-a-Number
Implemented functions

MPROOT for $x^{-1/n}$
MPEXP1 for $\exp(x) - 1$, MPEXP for $\exp(x)$
MPLNS for $\log(1 + x)$, MPLN for $\log(x)$
MPSIN, MPTAN, MPATAN, MPASIN,
MPPI for $\pi$, MPEUL for $\gamma$,
MPGAM for $\Gamma(x)$,
MPBERN for the Bernoulli numbers
MPEI, MPERF, MPERFC
MPBESJ
The Augment Interface

(with J. A. Hooper and J. M. Yohe)

MULTIPLE X, Y, Z

... 

X = Y + Z*EXP(X+1)/Y
Applications of MP


The constants were computed twice, once with base 10000 and 260 floating-point digits, and once with base 11701 and 250 digits (10000^{260} = 10^{1040}, 11701^{250} \approx 10^{1017}).
It is also impracticable to formally prove correctness of any nontrivial MP routines using present theorem-proving techniques.

In the future we hope to implement rounding options for more MP routines, and write a multiple-precision interval arithmetic package which uses MP and takes advantage of the directed rounding options.

A never-ending project is to implement multiple-precision versions of ever more special functions, and to improve the efficiency of those multiple-precision routines already implemented.
Visit to ANU, February 2007
Visit to ANU, February 2007
... to MPFR
Notations

MPFR uses GMP’s \texttt{mpn} layer for its internal representation

\textit{limb}: a GMP machine word (usually 32 or 64 bits)

For simplicity we will assume the number of bits in a limb is 64 in this talk.
Number Representation in MPFR

- 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 (NaN, $\pm \infty$, $\pm 0$)
- significand (array of $\lceil p/64 \rceil$ limbs), only defined for regular numbers (neither NaN, nor $\pm \infty$, nor $\pm 0$)

Most significant limbs/bits will be represented left in this talk. Regular numbers are normalized: the most significant bit of the most significant limb should be set.

Example: $x = 17$ with a precision of 10 bits is stored with a 6-bit limb as

```
<table>
<thead>
<tr>
<th>precision</th>
<th>sign</th>
<th>exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>+1</td>
<td>5</td>
</tr>
</tbody>
</table>
```

```
word 1: 100010
word 0: 000000
```
Round 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 at position \( p + 1 \).

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

The round and sticky bits are enough to get *correct rounding* for all rounding modes:

\[
\begin{array}{cccccc}
    r & s & \text{zero} & \text{nearest} & \text{away} \\
    0 & 0 & m & m & m \\
    0 & 1 & m & m & m + 1 \\
    1 & 0 & m & m + (m \mod 2) & m + 1 \\
    1 & 1 & m & m + 1 & m + 1 \\
\end{array}
\]
The mpfr_add function

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

- first check for singular values (NaN, ±Inf, ±0)
- if b and c have different signs, call mpfr_sub1
- if a, b, c have the same precision, call mpfr_add1sp
- otherwise call the generic code mpfr_add1 described in:
The mpfr_add1sp function

- if $p < 64$, call mpfr_add1sp1
- if $64 < p < 128$, call mpfr_add1sp2
- else execute the generic addition code for same precision

Note: $p = 64$ and $p = 128$ will use the generic code, thus should be avoided unless really needed.
The mpfr_add1sp1 function

Case 1, $e_b = e_c$:

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

\[
\begin{align*}
ap[0] &= \text{MPFR\_LIMB\_HIGHBIT} \mid ((bp[0] + cp[0]) \gg 1); \\
e_a &= e_b + 1; \\
r_b &= ap[0] \& (\text{MPFR\_LIMB\_ONE} \ll (sh - 1)); \\
ap[0] &= r_b; \\
sb &= 0; \\
\end{align*}
\]

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

Thus adding $bp[0]$ and $cp[0]$ will always produce a carry, and the exponent of $a$ will be $e_b + 1$. 
\[ b = \boxed{110100} \]

\[ c = \boxed{111000} \]

\[ \text{ap}[0] = \text{MPFR\_LIMB\_HIGHBIT} \mid ((\text{bp}[0] + \text{cp}[0]) \gg 1); \]
\[ \text{e}_a = \text{e}_b + 1; \]
\[ \text{rb} = \text{ap}[0] \& (\text{MPFR\_LIMB\_ONE} \ll (\text{sh} - 1)); \]
\[ \text{ap}[0] \text{ ^= } \text{rb}; \]
\[ \text{sb} = 0; \]

The sum might have up to \( p + 1 \) bits, but since \( p < 64 \) (\( p < 6 \) here), it fits on 64 bits.

\( \text{sh} \) is the number \( 64 - p \) of trailing bits, here \( 6 - p = 2 \).

The round bit is the \( (p + 1) \)-th bit of the addition, the sticky bit is always zero.

An overflow might happen, but no underflow.
The `mpfr_sub` function

The `mpfr_sub(a, b, c)` function 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 `mpfr_add1`
- if $b$ and $c$ have the same precision, call `mpfr_sub1sp`
- otherwise call the generic code `mpfr_sub1`
The mpfr_sub1sp function

- if \( p < 64 \), call mpfr_sub1sp1
- if \( 64 < p < 128 \), call mpfr_sub1sp2
- else execute the generic subtraction code for same precision

Note: \( p = 64 \) and \( p = 128 \) will use the generic code, thus should be avoided unless really needed.
The \texttt{mpfr\_sub1sp1} function

- if the exponents differ, swap $b$ and $c$ so that $e_b > e_c$
- case 1: $e_b = e_c$
- case 2: $e_b > e_c$
Case 1, $e_b = e_c$:

\[ b = \boxed{110100} \]

\[ c = \boxed{111000} \]

subtract $bp[0] - cp[0]$ and put the result in $ap[0]$, which is $bp[0] - cp[0] \mod 2^{64}$

if $ap[0] = 0$, then the result is zero

if $ap[0] > bp[0]$, then a borrow occurred thus $|c| > |b|$: change $ap[0]$ to $-ap[0]$ and change the sign of $a$

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

count the number of leading zeros in $ap[0]$, shift $ap[0]$ accordingly and decrease the exponent

in that case both the round bit and the sticky bit are zero

An underflow might happen, no overflow since $|a| \leq \max(|b|, |c|)$
The `mpfr_mul(a,b,c)` function

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

- if \( p_a < 64 \) and \( p_b, p_c \leq 64 \), call `mpfr_mul_1`
- if \( 64 < p_a < 128 \) and \( 64 < p_b, p_c \leq 128 \), call `mpfr_mul_2`
- else use the generic code
The mpfr_mul_1 function

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

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

\[ h \cdot 2^{64} + \ell \leftarrow bp[0] \cdot cp[0] \]

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

If \( h < 2^{63} \), shift \( h, \ell \) by 1 bit to the left, and decrease the exponent

The round bit is formed by the \((p + 1)\)-th bit of \( h \)

The sticky bit is formed by the remaining bits of \( h \), and those of \( \ell \)

Both underflow and overflow can happen

Warning: MPFR considers underflow after rounding (with an infinite exponent range)
The mpfr\_div(a,b,c) function

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

- if \( p_a < 64 \) and \( p_b, p_c \leq 64 \), call mpfr\_div\_1
- if \( 64 < p_a < 128 \) and \( 64 < p_b, p_c \leq 128 \), call mpfr\_div\_2
- else use the generic code
The mpfr_div_1 function

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

Assume \( p_a < 64 \) and \( p_b, p_c \leq 64 \)

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

Deal separately with the special case where the target precision is less than 32, and the divisor \( cp[0] \) has at most 32 bits. Then a single 64/32-bit division suffices. (Code used when dividing two binary32 numbers.)

General case: perform a 128/64 integer division, calling GMP's udiv_qrnnd_preinv routine. This yields a quotient of 64 bits, and a remainder, from which the round and sticky bit are deduced.
\[ bp[0] \cdot 2^{64} = q \cdot cp[0] + r \]

With `-enable-gmp-internals`, `udiv_qrnnd_preinv` uses GMP's `mpn_invert_limb` routine, which given \( 2^{63} \leq d < 2^{64} \), returns \( \lfloor (2^{128} - 1)/d - 2^{64} \rfloor \).

\[ i = \lfloor (2^{128} - 1)/cp[0] - 2^{64} \rfloor \]

\[ q \approx bp[0] + (i \cdot bp[0]/2^{64}) \]

Without `-enable-gmp-internals`, we let \( d = d_12^{32} + d_0 \), and perform two divisions by \( d_1 \) using its pseudo-inverse \( i = \lfloor (2^{64} - 1)/d_1 \rfloor \). This is slightly slower.
The `mpfr_sqrt(r, u)` function

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

- if \( p_r < 64 \) and \( p_u < 64 \), call `mpfr_sqrt1`
- if \( 64 < p_r < 128 \) and \( 64 < p_u \leq 128 \), call `mpfr_sqrt2`
- else use the generic code
The mpfr_sqrt1 function

Input: $2^{63} \leq u < 2^{64}$ representing a $p$-bit number with $p < 64$ (thus its least significant bit is 0)

- if the exponent of $u$ is odd, shift $u$ by one bit to the right

Now $2^{62} \leq u < 2^{64}$

- call mpn_sqrtrem2, a routine returning $r$ and $s$ such that

$$u \cdot 2^{64} = r^2 + s \quad \text{with } 0 \leq s \leq 2r$$

We have $2^{63} \leq r < 2^{64}$ and $0 \leq s < 2^{65}$, thus $s$ is represented by one 64-bit word and one bit.

- deduce the round bit from $r$, and the sticky bit from $s$ and the last bits of $r$ (if $p < 63$).
The mpfr_sqrtnrem2 function

Input:  \( u := u_3 \cdot 2^{192} + u_2 \cdot 2^{128} + u_1 \cdot 2^{64} + u_0 \) with \( 0 \leq u_j < 2^{64} \)

Output: \( r \) and \( s \) such that \( u = r^2 + s \) with \( u < (r+1)^2 \).

GMP provides a mpn_sqrtnrem function but it is slow.

mpfr_sqrtnrem2 works as follows:

• using a bipartite table reading the leading 12 = 4 + 4 + 4 bits of \( u \), obtain a 17-bit approximation of \( u^{-1/2} \) with about 9 correct bits

\[
\begin{align*}
u &= \underbrace{xxxx}_{a} \underbrace{yyyy}_{b} \underbrace{zzzz}_{c} \cdots \\
x_0 &= T_1[a, b] + T_2[b, c]
\end{align*}
\]
• using Newton’s iteration for the inverse square root, obtain a 32-bit approximation of $u^{-1/2}$ with about 19 correct bits

$$x_1 \approx x_0 + \frac{x}{2}(1 - ux_0^2)$$

• using Newton’s iteration for the inverse square root, obtain a 41-bit approximation $x$ of $u^{-1/2}$ with about 38 correct bits, ensuring $x_2 \leq u^{-1/2}$

$$x_2 \approx x_1 + \frac{x}{2}(1 - ux_1^2)$$

• use Karp-Markstein trick to deduce a 64-bit approximation $y'$ of $u^{1/2}$

$$y \approx ax_2, \quad y' \approx y + \frac{x_2}{2}(a - y^2)$$
MPFR 3.1.4 against MPF (from GMP 6.1.1)

bavette.loria.fr, Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz, running at 3.3Ghz, with GMP 6.1.1 and GCC 6.1.1.

<table>
<thead>
<tr>
<th>MPFR 3.1.4</th>
<th>MPF from GMP 6.1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits</td>
<td>limbs</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>113</td>
<td>113</td>
</tr>
<tr>
<td>mpfr_add</td>
<td>mpfr_add</td>
</tr>
<tr>
<td>37</td>
<td>49</td>
</tr>
<tr>
<td>44</td>
<td>49</td>
</tr>
<tr>
<td>48</td>
<td>45</td>
</tr>
<tr>
<td>mpfr_sub</td>
<td>mpfr_sub</td>
</tr>
<tr>
<td>44</td>
<td>52</td>
</tr>
<tr>
<td>50</td>
<td>52</td>
</tr>
<tr>
<td>56</td>
<td>48</td>
</tr>
<tr>
<td>mpfr_mul</td>
<td>mpfr_mul</td>
</tr>
<tr>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>44</td>
<td>43</td>
</tr>
<tr>
<td>59</td>
<td>46</td>
</tr>
<tr>
<td>mpfr_div</td>
<td>mpfr_div</td>
</tr>
<tr>
<td>115</td>
<td>81</td>
</tr>
<tr>
<td>116</td>
<td>81</td>
</tr>
<tr>
<td>131</td>
<td>146</td>
</tr>
<tr>
<td>mpfr_sqrt</td>
<td>mpfr_sqrt</td>
</tr>
<tr>
<td>152</td>
<td>236</td>
</tr>
<tr>
<td>153</td>
<td>234</td>
</tr>
<tr>
<td>244</td>
<td>339</td>
</tr>
</tbody>
</table>

Timings are in cycles.
MPFR 3.1.4 against MPFR 4.0-dev

MPFR 4.0-dev is configured with -enable-gmp-internals.

<table>
<thead>
<tr>
<th></th>
<th>MPFR 3.1.4</th>
<th></th>
<th></th>
<th></th>
<th>MPFR 4.0-dev</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bits</td>
<td>24</td>
<td>53</td>
<td>113</td>
<td>bits</td>
<td>24</td>
<td>53</td>
</tr>
<tr>
<td>mpfr_add</td>
<td></td>
<td>37</td>
<td>44</td>
<td>48</td>
<td>mpfr_add</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>mpfr_sub</td>
<td></td>
<td>44</td>
<td>50</td>
<td>56</td>
<td>mpfr_sub</td>
<td>29</td>
<td>31</td>
</tr>
<tr>
<td>mpfr_mul</td>
<td></td>
<td>42</td>
<td>44</td>
<td>59</td>
<td>mpfr_mul</td>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>mpfr_div</td>
<td></td>
<td>115</td>
<td>116</td>
<td>131</td>
<td>mpfr_div</td>
<td>48</td>
<td>57</td>
</tr>
<tr>
<td>mpfr_sqrt</td>
<td></td>
<td>152</td>
<td>153</td>
<td>244</td>
<td>mpfr_sqrt</td>
<td>48</td>
<td>72</td>
</tr>
</tbody>
</table>

Timings are in cycles.
MPFR 4.0-dev against MPF (from GMP 6.1.1)

MPFR is configured with `-enable-gmp-internals`.

<table>
<thead>
<tr>
<th></th>
<th>MPFR 4.0-dev</th>
<th></th>
<th>MPF from GMP 6.1.1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bits</td>
<td>limbs</td>
<td>bits</td>
<td>limbs</td>
</tr>
<tr>
<td>mpfr_add</td>
<td>24 53 113</td>
<td></td>
<td>mpfr_add</td>
<td>24 53 113</td>
</tr>
<tr>
<td>mpfr_sub</td>
<td>29 31 32</td>
<td></td>
<td>mpfr_sub</td>
<td>24 53 113</td>
</tr>
<tr>
<td>mpfr_mul</td>
<td>22 21 33</td>
<td></td>
<td>mpfr_mul</td>
<td>24 53 113</td>
</tr>
<tr>
<td>mpfr_div</td>
<td>48 57 87</td>
<td></td>
<td>mpfr_div</td>
<td>24 53 113</td>
</tr>
<tr>
<td>mpfr_sqrt</td>
<td>48 72 128</td>
<td></td>
<td>mpfr_sqrt</td>
<td>24 53 113</td>
</tr>
</tbody>
</table>

Timings are in cycles.