Outline

Introduction and context

The Table Maker’s dilemma

One algorithm, many variants

Results

Bonus: a floating-point in, fixed-point out variant

Conclusions
Preparing 2017, international year of the logarithm

John Napier (aka Neper), 1550-1617

- popularized the use of the point in decimal notation
John Napier (aka Neper), 1550-1617

- popularized the use of the point in decimal notation
- *Mirifici Logarithmorum Canonis Descriptio* (1614)
Preparing 2017, international year of the logarithm

John Napier (aka Neper), 1550-1617
- popularized the use of the point in decimal notation
- *Mirifici Logarithmorum Canonis Descriptio* (1614)

Celebrate a very specific year:
- 400th anniversary of Napier’s death
- 6th logarithmic anniversary of the 1614 publication

... with three amazing presentations this morning,
now doubt they will trigger many others.
This talk is also about hardware and C

1960  1980  2000

IEEE-754  mainstream floating-point

32-bits  mainstream integer

An experiment
Implementing the floating-point logarithm function using only integer arithmetic for performance (previous work motivated by lack of FP hardware)
This talk is also about hardware and C

1960     1980     2000

IEEE-754 mainstream floating-point
32-bits mainstream integer
64-bits

An experiment
Implementing the floating-point logarithm function using only integer arithmetic for performance (previous work motivated by lack of FP hardware)

J. Le Maire, F. de Dinechin and J.-M. Muller
Computing correctly rounded logarithm with fixed-point operations
This talk is also about hardware and C

An experiment
Implementing the floating-point logarithm function
- using only integer arithmetic
- for performance

(previous work motivated by lack of FP hardware)
64-bit floating-point, but only 52-bit precision
  - if you can predict the value of the exponent, exponent bits are wasted bits.
• 64-bit floating-point, but only 52-bit precision
  • if you can predict the value of the exponent, exponent bits are wasted bits.
• modern 64-bit machines offer all sort of useful integer instructions
  • addition
  • multiplication $64 \times 64 \rightarrow 128$ (\textit{mulq})
  • count leading zeroes, shifts (\textit{lzcnt, bsr})
Integer better than floating-point?

- 64-bit floating-point, but only 52-bit precision
  - if you can predict the value of the exponent, exponent bits are wasted bits.
- modern 64-bit machines offer all sort of useful integer instructions
  - addition
  - multiplication $64 \times 64 \rightarrow 128$ \((mulq)\)
  - count leading zeroes, shifts \((lzcnt, bsr)\)
- most operations are faster on integers, especially addition
  (which more or less defines the processor cycle time)
Integer better than floating-point?

- 64-bit floating-point, but only 52-bit precision
  - if you can predict the value of the exponent, exponent bits are wasted bits.
- modern 64-bit machines offer all sort of useful integer instructions
  - addition
  - multiplication 64x64 → 128 \((\text{mulq})\)
  - count leading zeroes, shifts \((\text{lzcnt, bsr})\)
- most operations are faster on integers, especially addition
  (which more or less defines the processor cycle time)
- small multiprecision out of the box:
  mainstream compilers \((\text{gcc, clang, icc})\) support int_128
  - addition 128x128 → 128 \((\text{add, adc})\)
  - shift on two registers \((\text{shld, shrd})\)
Integer better than floating-point?

- 64-bit floating-point, but only 52-bit precision
  - if you can predict the value of the exponent, exponent bits are wasted bits.
- modern 64-bit machines offer all sort of useful integer instructions
  - addition
  - multiplication $64 \times 64 \rightarrow 128$ \textit{(mulq)}
  - count leading zeroes, shifts \textit{(lzcnt, bsr)}
- most operations are faster on integers, especially addition
  (which more or less defines the processor cycle time)
- small multiprecision out of the box:
  - mainstream compilers \textit{(gcc, clang, icc)} support \texttt{int\_128}
    - addition $128 \times 128 \rightarrow 128$ \textit{(add, adc)}
    - shift on two registers \textit{(shld, shrd)}

Caveat: integer SIMD/vector support still lagging behind FP
(no vector multiplication)
Logarithm, the mathematical version

\[ \ln(a \times b) = \ln(a) + \ln(b) \]

Taylor: for \( x \) small, \( \ln(1+x) \approx x - \frac{x^2}{2} + \frac{x^3}{3} + \ldots \)
Logarithm, the mathematical version

\[ \ln(a \times b) = \ln(a) + \ln(b) \]

\[ y = \ln(x) \]
Logarithm, the mathematical version

- \( \ln(a \times b) = \ln(a) + \ln(b) \)
- \( \ln(b^a) = a \times \ln(b) \)

\[
y = \ln(x)
\]

J. Le Maire, F. de Dinechin and J.-M. Muller
Computing correctly rounded logarithm with fixed-point operations
Logarithm, the mathematical version

- \( \ln(a \times b) = \ln(a) + \ln(b) \)
- \( \ln(b^a) = a \times \ln(b) \)
- Taylor: for \( x \) small, \( \ln(1 + x) \approx x - x^2/2 + x^3/3 \ldots \)
The natural logarithm is called log
(you will also find log2 and log10 and a few others)

\[ y = \ln(x) \]

Range: \( \forall x \in \mathbb{F}_{64} \quad \log(x) \in [-745, 710] \)

- looks like a waste of exponent bits...
The natural logarithm is called \( \log \) (you will also find \( \log_2 \) and \( \log_{10} \) and a few others)

\[
y = \ln(x)
\]

- **Range:** \( \forall x \in F_{64} \quad \log(x) \in [-745, 710] \)
  - looks like a waste of exponent bits...
- **Rounding**
  - **Recommended:** \( \forall x \in F_{64} \quad \log(x) = \circ(\ln(x)) \)
  - In practice: implementing this definition difficult and expensive, due to the Table Maker's dilemma.
Introduction and context

The Table Maker’s dilemma

One algorithm, many variants

Results

Bonus: a floating-point in, fixed-point out variant

Conclusions
The first digital signature algorithm

I want 12 significant digits
I have an approximation scheme that provides 14
digits

\[ y = \log(x) \pm 10^{-14} \]

'Usually' that's enough to round

\[ y = x, \quad x \pm 10^{-14} \]

Dilemma when

\[ y = x, \quad x \pm 10^{-14} \]

The first table-makers rounded these cases randomly,
and recorded them to confound copiers.

---

**LOGARITHMICA.**

<table>
<thead>
<tr>
<th>Table of logarithms</th>
<th>0.0001</th>
<th>0.0002</th>
<th>0.0003</th>
<th>0.0004</th>
<th>0.0005</th>
<th>0.0006</th>
<th>0.0007</th>
<th>0.0008</th>
<th>0.0009</th>
<th>0.0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.001</td>
<td>1.0000</td>
<td>1.0001</td>
<td>1.0002</td>
<td>1.0003</td>
<td>1.0004</td>
<td>1.0005</td>
<td>1.0006</td>
<td>1.0007</td>
<td>1.0008</td>
<td>1.0009</td>
</tr>
<tr>
<td>0.002</td>
<td>2.0000</td>
<td>2.0001</td>
<td>2.0002</td>
<td>2.0003</td>
<td>2.0004</td>
<td>2.0005</td>
<td>2.0006</td>
<td>2.0007</td>
<td>2.0008</td>
<td>2.0009</td>
</tr>
<tr>
<td>0.003</td>
<td>3.0000</td>
<td>3.0001</td>
<td>3.0002</td>
<td>3.0003</td>
<td>3.0004</td>
<td>3.0005</td>
<td>3.0006</td>
<td>3.0007</td>
<td>3.0008</td>
<td>3.0009</td>
</tr>
<tr>
<td>0.004</td>
<td>4.0000</td>
<td>4.0001</td>
<td>4.0002</td>
<td>4.0003</td>
<td>4.0004</td>
<td>4.0005</td>
<td>4.0006</td>
<td>4.0007</td>
<td>4.0008</td>
<td>4.0009</td>
</tr>
<tr>
<td>0.005</td>
<td>5.0000</td>
<td>5.0001</td>
<td>5.0002</td>
<td>5.0003</td>
<td>5.0004</td>
<td>5.0005</td>
<td>5.0006</td>
<td>5.0007</td>
<td>5.0008</td>
<td>5.0009</td>
</tr>
</tbody>
</table>

---

J. Le Maire, F. de Dinechin and J.-M. Muller

Computing correctly rounded logarithm with fixed-point operations
The first digital signature algorithm

I want 12 significant digits

I have an approximation scheme that provides 14 digits

\[ y = \log(x) \pm 10^{-14} \]

"Usually" that's enough to round

Dilemma when

The first table-makers rounded these cases randomly, and recorded them to confound copiers.

J. Le Maire, F. de Dinechin and J.-M. Muller

Computing correctly rounded logarithm with fixed-point operations
I want 12 significant digits

I have an approximation scheme that provides 14 digits
I want 12 significant digits
I have an approximation scheme that provides 14 digits
or,

\[ y = \log(x) \pm 10^{-14} \]

---

| E | \# \begin{array}{rrr}
| 0.0001 | 0.0000 | 0.0000 |
| 0.0002 | 0.0000 | 0.0000 |
| 0.0003 | 0.0000 | 0.0000 |
| 0.0004 | 0.0000 | 0.0000 |
| 0.0005 | 0.0000 | 0.0000 |
| 0.0006 | 0.0000 | 0.0000 |
| 0.0007 | 0.0000 | 0.0000 |
| 0.0008 | 0.0000 | 0.0000 |
| 0.0009 | 0.0000 | 0.0000 |
| 0.0010 | 0.0000 | 0.0000 |
| 0.0011 | 0.0000 | 0.0000 |
| 0.0012 | 0.0000 | 0.0000 |
| 0.0013 | 0.0000 | 0.0000 |
| 0.0014 | 0.0000 | 0.0000 |
| 0.0015 | 0.0000 | 0.0000 |
| 0.0016 | 0.0000 | 0.0000 |
| 0.0017 | 0.0000 | 0.0000 |
| 0.0018 | 0.0000 | 0.0000 |
| 0.0019 | 0.0000 | 0.0000 |
| 0.0020 | 0.0000 | 0.0000 |
| 0.0021 | 0.0000 | 0.0000 |
| 0.0022 | 0.0000 | 0.0000 |
| 0.0023 | 0.0000 | 0.0000 |
| 0.0024 | 0.0000 | 0.0000 |
| 0.0025 | 0.0000 | 0.0000 |
| 0.0026 | 0.0000 | 0.0000 |
| 0.0027 | 0.0000 | 0.0000 |
| 0.0028 | 0.0000 | 0.0000 |
| 0.0029 | 0.0000 | 0.0000 |
| 0.0030 | 0.0000 | 0.0000 |
| 0.0031 | 0.0000 | 0.0000 |
| 0.0032 | 0.0000 | 0.0000 |
| 0.0033 | 0.0000 | 0.0000 |
| 0.0034 | 0.0000 | 0.0000 |
| 0.0035 | 0.0000 | 0.0000 |
| 0.0036 | 0.0000 | 0.0000 |
| 0.0037 | 0.0000 | 0.0000 |
| 0.0038 | 0.0000 | 0.0000 |
| 0.0039 | 0.0000 | 0.0000 |
| 0.0040 | 0.0000 | 0.0000 |
| 0.0041 | 0.0000 | 0.0000 |
| 0.0042 | 0.0000 | 0.0000 |
| 0.0043 | 0.0000 | 0.0000 |
| 0.0044 | 0.0000 | 0.0000 |
| 0.0045 | 0.0000 | 0.0000 |
| 0.0046 | 0.0000 | 0.0000 |
| 0.0047 | 0.0000 | 0.0000 |
| 0.0048 | 0.0000 | 0.0000 |
| 0.0049 | 0.0000 | 0.0000 |
| 0.0050 | 0.0000 | 0.0000 |
| 0.0051 | 0.0000 | 0.0000 |
| 0.0052 | 0.0000 | 0.0000 |
| 0.0053 | 0.0000 | 0.0000 |
| 0.0054 | 0.0000 | 0.0000 |
| 0.0055 | 0.0000 | 0.0000 |
| 0.0056 | 0.0000 | 0.0000 |
| 0.0057 | 0.0000 | 0.0000 |
| 0.0058 | 0.0000 | 0.0000 |
| 0.0059 | 0.0000 | 0.0000 |
| 0.0060 | 0.0000 | 0.0000 |
| 0.0061 | 0.0000 | 0.0000 |
| 0.0062 | 0.0000 | 0.0000 |
| 0.0063 | 0.0000 | 0.0000 |
| 0.0064 | 0.0000 | 0.0000 |
| 0.0065 | 0.0000 | 0.0000 |
| 0.0066 | 0.0000 | 0.0000 |
| 0.0067 | 0.0000 | 0.0000 |
| 0.0068 | 0.0000 | 0.0000 |
| 0.0069 | 0.0000 | 0.0000 |
| 0.0070 | 0.0000 | 0.0000 |
| 0.0071 | 0.0000 | 0.0000 |
| 0.0072 | 0.0000 | 0.0000 |
| 0.0073 | 0.0000 | 0.0000 |
| 0.0074 | 0.0000 | 0.0000 |
| 0.0075 | 0.0000 | 0.0000 |
| 0.0076 | 0.0000 | 0.0000 |
| 0.0077 | 0.0000 | 0.0000 |
| 0.0078 | 0.0000 | 0.0000 |
| 0.0079 | 0.0000 | 0.0000 |
| 0.0080 | 0.0000 | 0.0000 |
| 0.0081 | 0.0000 | 0.0000 |
| 0.0082 | 0.0000 | 0.0000 |
| 0.0083 | 0.0000 | 0.0000 |
| 0.0084 | 0.0000 | 0.0000 |
| 0.0085 | 0.0000 | 0.0000 |
| 0.0086 | 0.0000 | 0.0000 |
| 0.0087 | 0.0000 | 0.0000 |
| 0.0088 | 0.0000 | 0.0000 |
| 0.0089 | 0.0000 | 0.0000 |
| 0.0090 | 0.0000 | 0.0000 |
| 0.0091 | 0.0000 | 0.0000 |
| 0.0092 | 0.0000 | 0.0000 |
| 0.0093 | 0.0000 | 0.0000 |
| 0.0094 | 0.0000 | 0.0000 |
| 0.0095 | 0.0000 | 0.0000 |
| 0.0096 | 0.0000 | 0.0000 |
| 0.0097 | 0.0000 | 0.0000 |
| 0.0098 | 0.0000 | 0.0000 |
| 0.0099 | 0.0000 | 0.0000 |
| 0.0100 | 0.0000 | 0.0000 |

---

J. Le Maire, F. de Dinechin and J.-M. Muller
Computing correctly rounded logarithm with fixed-point operations
The first digital signature algorithm

I want 12 significant digits
I have an approximation scheme that provides 14 digits
or,
\[ y = \log(x) \pm 10^{-14} \]
"Usually" that's enough to round
\[ y = x, \text{xxxxxxxxxxx}17 \pm 10^{-14} \]
y = x, xxxxxxxxxxxxx83 \pm 10^{-14}
The first digital signature algorithm

I want 12 significant digits

I have an approximation scheme that provides 14 digits

or,

$$y = \log(x) \pm 10^{-14}$$

“Usually” that’s enough to round

$$y = x, xxxxxxxxxxxx17 \pm 10^{-14}$$

$$y = x, xxxxxxxxxxxx83 \pm 10^{-14}$$

Dilemma when

$$y = x, xxxxxxxxxxxx50 \pm 10^{-14}$$
The first digital signature algorithm

I want 12 significant digits
I have an approximation scheme that provides 14 digits
or,
\[ y = \log(x) \pm 10^{-14} \]
"Usually" that’s enough to round
\[ y = x, \quad \pm 10^{-14} \]

Dilemma when
\[ y = x, \quad \pm 10^{-14} \]

The first table-makers rounded these cases randomly, and recorded them to confound copiers.

---

J. Le Maire, F. de Dinechin and J.-M. Muller

Computing correctly rounded logarithm with fixed-point operations

9
Solving the Table Maker’s dilemma

two consecutive floating-point numbers

real numbers
Solving the Table Maker’s dilemma

Real numbers

Two consecutive floating-point numbers

Computed logarithm, with error margin

\[ y = x, \underbrace{xxxxxxx}_{17} \pm 10^{-14} \]

Easy to round
Solving the Table Maker’s dilemma

real numbers

two consecutive floating-point numbers

computed logarithm, with error margin

\[ y = x, \quad xxxxxxxxxxxxxx50 \pm 10^{-14} \]

Difficult to round
Solving the Table Maker's dilemma

real numbers

two consecutive floating-point numbers

computed logarithm, with error margin

$y = x, xxxxxxxxxxxxx4996 \pm 10^{-16}$
Computing more accurately solves it
Solving the Table Maker’s dilemma

There is a finite number (2^64) of floating-point numbers. One of them is the worst to round. Muller and Lefèvre computed that it requires an accuracy of $2^{-113}$: evaluating the log to this accuracy enables correct rounding but we don't need this accuracy for most cases (and it is more expensive to compute).
\forall x \in \mathbb{F}, \ln(x) \text{ is transcendental}

- There is a finite number \((2^{64})\) of floating-point numbers.
\[ \forall x \in \mathbb{F}, \ln(x) \text{ is transcendental} \]

\[ \text{There is a finite number (}2^{64}\text{) of floating-point numbers.} \]

\[ \text{One of them is the worst to round} \]
Solving the Table Maker's dilemma

- $\forall x \in \mathbb{F}, \ln(x)$ is transcendental
- There is a finite number ($2^{64}$) of floating-point numbers.
- One of them is the worst to round
- Muller and Lefèvre computed that it requires an accuracy of $2^{-113}$: evaluating the log to this accuracy enables correct rounding.
Solving the Table Maker's dilemma

∀x ∈ ℝ, ln(x) is transcendental

There is a finite number (2^{64}) of floating-point numbers.

One of them is the worst to round

Muller and Lefèvre computed that it requires an accuracy of 2^{−113}: evaluating the log to this accuracy enables correct rounding

but we don’t need this accuracy for most cases

(and it is more expensive to compute)
∀x ∈ F, ln(x) is transcendental
- There is a finite number (2^{64}) of floating-point numbers.
- One of them is the worst to round
- Muller and Lefèvre computed that it requires an accuracy of 2^{-113}:
  evaluating the log to this accuracy enables correct rounding
- but we don’t need this accuracy for most cases
  (and it is more expensive to compute)
CRLibm refinement of Ziv’s technique:

- First step: quick-and-dirty evaluation of $\ln(x)$
  (just accurate enough to ensure correct rounding in most cases)
- test if rounding can be decided
- if not (rarely), recompute $\ln(x)$ with the worst-case accuracy
CRLibm refinement of Ziv’s technique:

- First step: quick-and-dirty evaluation of $\ln(x)$
  (just accurate enough to ensure correct rounding in most cases)
- test if rounding can be decided
- if not (rarely), recompute $\ln(x)$ with the worst-case accuracy

Trade-off between first and second steps:

$$Mean Time = Time(1st \ step) + Pr[need \ 2nd \ step] \cdot Time(2nd \ step)$$
CRLibm refinement of Ziv’s technique:
- First step: quick-and-dirty evaluation of \(\ln(x)\)
  (just accurate enough to ensure correct rounding in most cases)
- test if rounding can be decided
- if not (rarely), recompute \(\ln(x)\) with the worst-case accuracy

Trade-off between first and second steps:

\[
\text{MeanTime} = \text{Time(1st step)} + \Pr[\text{need 2nd step}] \cdot \text{Time(2nd step)}
\]

Best so far: \(\text{Time(2nd step)} \approx 10 \times \text{Time(1st step)}\)
In this work we improve this to a factor 2.
Introduction and context

The Table Maker’s dilemma

One algorithm, many variants

Results

Bonus: a floating-point in, fixed-point out variant

Conclusions
The big picture

1. Filter special cases (negative numbers, \(\infty\), ...)
2. Argument range reduction
3. Polynomial approximation
4. Solution reconstruction
5. Error evaluation and rounding test
6. If more accuracy needed:
   - Rerun the steps 3 and 4 with the worst-case accuracy.
IEEE 754 floating-point

The IEEE 754 floating-point format consists of a sign bit, a 11-bit exponent, and a 52-bit fraction. The value represented is given by:

\[ (-1)^s \cdot 2^E \cdot (1 + x) \]

Where:
- \( s \) is the sign bit (1 for negative, 0 for positive)
- \( E \) is the exponent (11 bits, with a bias of 1023)
- \( x \) is the fraction (52 bits, in the range \([0, 1)\))
IEEE 754 floating-point

Value represented:

\[ (-1)^s \cdot 2^E \cdot (1 + x) \]

Special cases \((\pm \infty, 0, NaN)\) encoded in special values of the exponent field
/* reinterpreted x to manipulate its bits more easily */
uint64_t xbits = ((union { double d; uint64_t u; }){x}).u;
int xe = xbits >> 52;

/* filter the special cases: !(x is normalized and 0 < x < +Inf) */
if (0x7FEu <= (unsigned)xe - 1u) {
    /* x = +0: raise a DivideByZero, return -Inf */
    if (((xbits & ~(1ull << 63)) == 0) return -1.0/0.0;
    /* x < 0.0: raise a InvalidOperation, return a qNaN */
    if (((xbits & (1ull << 63)) != 0) return (x-x)/0;
    /* x = qNaN: return a qNaN
     x = sNaN: raise a InvalidOperation, return a qNaN
     x = +Inf: return +Inf */
    if (xe != 0) return x+x;
    /* x subnormal: change x to a normalized number */
    else {
        int u = clz64(xbits) - 12;
        xbits <<= u + 1;
        xe -= u;
    }
} /* X = 2^xe * (xbits/2^52) */
xe -= 1023;
xbits = (xbits & 0xFFFFFFFFFFFFFull) + (UINT64_C(1) << 52);
First argument range reduction

\[ \text{input} = 2^E \cdot (1 + x) \]
\[ \ln(\text{input}) = E \cdot \ln(2) + \ln(1 + x) \]
First argument range reduction

\[
\text{input} = 2^E \cdot (1 + x) \\
\ln(\text{input}) = E \cdot \ln(2) + \ln(1 + x)
\]

Evaluation algorithm:
- approximate \( \ln(1 + x) \) with a polynomial \( p(x) \) 
  \hspace{1cm} \text{degree needed: at least 26}
- evaluate \( E \cdot \ln(2) \)
- add both terms
Tang’s range reduction

\[ 1 + x: \quad \frac{1}{2^{52}} \text{ fractional part on 52 bits} \]

\[ \text{inv}_x: \quad 0, \ldots, 1 \]

- A table, addressed by the \( x_1 \) most significand bits of \( x \), stores

\[ \text{inv}_x \approx \frac{1}{1 + x} \quad \text{and} \quad \ln\left(\text{inv}_x\right) \]
A table, addressed by the $x_1$ most significand bits of $x$, stores

$$inv_x \approx \frac{1}{1 + x} \quad \text{and} \quad \ln(inv_x)$$

As $inv_x \cdot (1 + x) \approx 1$, define

$$inv_x \cdot (1 + x) = 1 + y$$
A table, addressed by the $x_1$ most significand bits of $x$, stores $\text{inv}_x \approx \frac{1}{1+x}$ and $\ln(\text{inv}_x)$.

As $\text{inv}_x \cdot (1 + x) \approx 1$, define $\text{inv}_x \cdot (1 + x) = 1 + y$.

Then

$$\ln(1 + x) = \ln(1 + y) - \ln(\text{inv}_x)$$
Tang's range reduction algorithm

1 + x: fractional part on 52 bits

$\text{inv}_x$: 0

1 + y: fractional part on 64 bits plus 6 implicit zeros

- Extract the index $x_1$
- Read, from a table addressed by $x_1$, both $\text{inv}_x$ and $\ln(\text{inv}_x)$
- compute $y = \text{inv}_x \cdot (1 + x) - 1$ (exactly)
- approximate $\ln(1 + y)$ with a polynomial $p(y)$

$$\text{Degree needed: 8}$$

- add it all:

$$\ln(\text{input}) \approx E \cdot \ln(2) + p(y) - \ln(\text{inv}_x)$$
Here integers are better than floating-point

1 + x:

\[ 1 \text{ fractional part on 52 bits} \]

\[ \times_{1} \]

\[ \text{fractional part on 18 bits} \]

\[ 1 + y: \]

\[ 1000000 \text{ fractional part on 64 bits plus 6 implicit zeros} \]

With a 53-bit \( 1 + x \) we can tabulate \( \text{inv}_x \) on 18 bits:

- the exact product would need 71 bits
- but we can predict the 7 leading bits
- ... so we can let them overflow quietly and use a \( 64 \times 64 \rightarrow 64 \) multiplication.
There are reciprocal approximation instructions in most recent processors, including this pentium.

Computing $y = \text{inv}_x \cdot (1 + x) - 1$ exactly requires an FMA, or double-extended, or a bit of double-FP arithmetic.
Two levels of Tang reduction

\[1 + x: \quad \frac{x_1}{1} \quad \text{fractional part on 52 bits}\]

\[\text{inv}_x: \quad 0. \quad \text{fractional part on 8 bits}\]

\[1 + y: \quad 100000 \quad \text{fractional part on 60 bits including 5 zeros}\]

\[\text{inv}_y: \quad 0. \quad \text{fractional part on 15 bits}\]

\[1 + z: \quad 100000000000 \quad \text{fractional part on 64 bits plus 11 implicit zeros}\]

\[x \in [0, 1)\]
\[y \in [0, 2^{-5.41503})\]
\[z \in [0, 2^{-11.8262})\]

\[x_1 \text{ takes 64 different values}\]
\[y_1 \text{ takes 96 different values}\]

Again, the whole reduction of \(x\) to \(z\) is \text{computed exactly} in 64-bit int.
/* $X = 2^{xe} \times (1/R) \times Y,$
    with $Y = y/2^{(52 + \text{ARG}_1\text{SIZE})}$
    and $1/R = \text{argReduc1}[ri].val/2^{\text{ARG}_1\text{SIZE}}$ */
uint8_t ri = (xbits >> (52 - \text{ARG}_1\text{PREC})) - (1u << \text{ARG}_1\text{PREC});
uint64_t y = \text{ARG}_1\text{GETVALUE}(ri) \times xbits;

/* $Y = (1/S) \times (1 + dZ),$ 
    with $dZ = dz/2^{(52 + \text{ARG}_1\text{SIZE} + \text{ARG}_2\text{SIZE})}$ 
    and $1/S = \text{argReduc2}[si].val/2^{\text{ARG}_2\text{SIZE}}$ */
uint8_t si = (y >> (52 + \text{ARG}_1\text{SIZE} - \text{ARG}_2\text{PREC})) - (1u << \text{ARG}_2\text{PREC});
uint64_t dz = \text{ARG}_2\text{GETVALUE}(si) \times y;
// the integer part of the fixed-point is removed by overflow
Why stop at two levels of reduction?

Answer is: diminishing return.

For a target accuracy of $2^{-60}$:

<table>
<thead>
<tr>
<th></th>
<th>interval of $x$</th>
<th>degree needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>No reduction</td>
<td>$[-1/2, 1/2]$</td>
<td>29</td>
</tr>
<tr>
<td>1 level</td>
<td>$[-2^{-7}, 2^{-7}]$</td>
<td>7</td>
</tr>
<tr>
<td>2 levels</td>
<td>$[-2^{-12}, 2^{-12}]$</td>
<td>4</td>
</tr>
<tr>
<td>3 levels</td>
<td>$[-2^{-18}, 2^{-18}]$</td>
<td>3</td>
</tr>
</tbody>
</table>

Adding more levels will cost more operations than it saves...
Parenthesis: hardware-oriented algorithms

I have been strongly encouraged to Alt-Tab to other irrelevant slides...

Arith 2007 “Return of the hardware elementary function”

- Iterate on the same range reduction
- Stop as soon as Taylor at order 2 is good enough:
  \[ p(z) = z - z^2/2 \]
  because it is very easy to compute
- Build ad-hoc rectangular multipliers
- No need to tabulate \( 1/(1 + x_i) \) when \( x_i \) is small enough.
Back to our business.
We want to approximate $\log(1 + z)$ on an interval around 0. Use the (now standard) tool set to obtain it.

- **Sollya:**
  - finds a machine-efficient polynomial $P(z)$
  - computes a safe bound on the approximation error $P(z) - \ln(1 + z)$

- **Gappa:** bounds the accumulation of rounding errors when evaluating $P(z)$ in C

We obtain a Coq proof of the error:

computed approximation of $\ln(1 + z)$, with error margin
Fixed-point means: explicit shifts

/* Polynomial approximation of \( \log(1+Z)/Z \approx P(Z) \), and evaluate \( Z*P(Z) \) */

uint64_t p = UINT64_C(0xfffffffffffffffff4)
- (highmul(dz, UINT64_C(0x7fffffffff091895))
  - (highmul(dz, UINT64_C(0x55555509230f34c))
    - (highmul(dz,UINT64_C(0x3ff8f2ad563f0e19))
      )>>IMPLICIT_ZEROS)
    )>>IMPLICIT_ZEROS)
  )>>IMPLICIT_ZEROS);

uint128_t zpzpart = fullmul(dz, p);

Note that some of the shifts are inside the constants
Reconstructing the solution

\[ \text{input} = 2^e \cdot (1 + x) \]
Reconstructing the solution

\[ \text{input} = 2^e \cdot \frac{1}{\text{inv}_x} \cdot (1 + y) \]
Reconstructing the solution

\[ \text{input} = 2^e \cdot \frac{1}{\text{inv}_x} \cdot \frac{1}{\text{inv}_y} \cdot (1 + z) \]
Reconstructing the solution

\[ \text{input} = 2^e \cdot \frac{1}{\text{inv}_x} \cdot \frac{1}{\text{inv}_y} \cdot (1 + z) \]

\[ \ln(\text{input}) = e \cdot \ln(2) + \ln(\text{inv}_x^{-1}) + \ln(\text{inv}_y^{-1}) + \ln(1 + z) \]
Reconstructing the solution

\[ input = 2^e \cdot \frac{1}{\text{inv}_x} \cdot \frac{1}{\text{inv}_y} \cdot (1 + z) \]

\[ \ln(input) = e \cdot \ln(2) + \ln(\text{inv}_x^{-1}) + \ln(\text{inv}_y^{-1}) + \ln(1 + z) \]

- \( e \cdot \ln(2) \):
- \( \ln(\text{inv}_x^{-1}) \):
- \( \ln(\text{inv}_y^{-1}) \):
- \( P(z) \approx \ln(1 + z) \):
- Sum:

“If we can predict the exponents, exponent bits are wasted bits”
Reconstructing the solution

\[ \text{input} = 2^e \cdot \frac{1}{\text{inv}_x} \cdot \frac{1}{\text{inv}_y} \cdot (1 + z) \]

\[
\ln(\text{input}) = e \cdot \ln(2) + \ln(\text{inv}_x^{-1}) + \ln(\text{inv}_y^{-1}) + \ln(1 + z)
\]

- \( e \cdot \ln(2) \):
- \( \ln(\text{inv}_x^{-1}) \):
- \( \ln(\text{inv}_y^{-1}) \):
- \( P(z) \approx \ln(1 + z) \):
- sum:

"If we can predict the exponents, exponent bits are wasted bits"
Reconstructing the solution

\[ \text{input} = 2^e \cdot \frac{1}{\text{inv}_x} \cdot \frac{1}{\text{inv}_y} \cdot (1 + z) \]

\[ \ln(\text{input}) = e \cdot \ln(2) + \ln(\text{inv}_x^{-1}) + \ln(\text{inv}_y^{-1}) + \ln(1 + z) \]

“If we can predict the exponents, exponent bits are wasted bits”
Reconstructing the solution

\[ \text{input} = 2^e \cdot \frac{1}{\text{inv}_x} \cdot \frac{1}{\text{inv}_y} \cdot (1 + z) \]

\[ \ln(\text{input}) = e \cdot \ln(2) + \ln(\text{inv}_x^{-1}) + \ln(\text{inv}_y^{-1}) + \ln(1 + z) \]

<table>
<thead>
<tr>
<th>e \cdot \ln(2):</th>
</tr>
</thead>
<tbody>
<tr>
<td>-53</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\ln(\text{inv}_x^{-1}):</th>
</tr>
</thead>
<tbody>
<tr>
<td>-11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\ln(\text{inv}_y^{-1}):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P(z) \approx \ln(1 + z):</th>
</tr>
</thead>
<tbody>
<tr>
<td>-117</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sum:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-117</td>
</tr>
</tbody>
</table>

“If we can predict the exponents, exponent bits are wasted bits”
Now it really gets ugly

```c
/* Compute part of the result that don't depend on Z
   (xe * log(2) + log(1/Ri) + log(1/Si)) */
uint128_t cstpart =
    fullimul(xe, log2fw_mid)
+ UINT128((int64_t)xextlog2fw_high, 0) // no full mul here
+ UINT128(argReduc1[ri].log_hi, argReduc1[ri].log_mid)
+ UINT128(argReduc2[si].log_hi, argReduc2[si].log_mid);

/* Assemble the two parts, compute the sign, mantissa and exponent */
uint128_t longres = cstpart + (zpzpart >> (11 + IMPLICIT_ZEROS));
uint64_t sign = - (HI(longres) >> 63);  // sign is 0 if result > 0
// if sign != 0, this is longres = ~longres: it approximate the abs value
// to avoid the approximation, do: longres = ((int64_t)sign + longres) ^ UINT128(sign, sign);
longres ^= UINT128(sign, sign);
int u = clz64(HI(longres)) + 1;
int exponent = 11 - u;
uint64_t mantissa = HI(longres << u);
```
\[ e \cdot \ln(2): \]
\[ \ln(\text{inv}_x^{-1}): \]
\[ \ln(\text{inv}_y^{-1}): \]
\[ P(z) \approx \ln(1 + z): \]
\[ \text{sum:} \]
\[ \sum |e| \cdot 2^{-11} - 117 \]
\[ e \cdot \ln(2): \]

\[ \ln(\text{inv}_x^{-1}): \]

\[ \ln(\text{inv}_y^{-1}): \]

\[ P(z) \approx \ln(1 + z): \]

\[ \text{sum}: \]

\[ \epsilon < (|e|) \cdot 2^{-117} \]
$e \cdot \ln(2)$:

$\ln(\text{inv}_x^{-1})$:

$\ln(\text{inv}_y^{-1})$:

$P(z) \approx \ln(1 + z)$:

$\epsilon < (|e| + 1 + 1) \cdot 2^{-117}$
\[ \epsilon < \left( |e| + 1 + 1 + P(z) \cdot 2^{-55} \right) \cdot 2^{-117} \]
Simple technique: compute the two bounds of the interval, and see if they round to the same mantissa

(two additions, a xor and a shift)
Rounding test

Simple technique: compute the two bounds of the interval, and see if they round to the same mantissa

(two additions, a xor and a shift)

real numbers
Simple technique: compute the two bounds of the interval, and see if they round to the same mantissa

(two additions, a xor and a shift)

For comparison, the proof of the floating-point-based rounding test (invented by Ziv and used in CRLibm) is an 18-page paper that took 20 years to publish...
/* Compute the maximal absolute error (aligned with longres) */

If result \( (1 + \text{maxRelErr}) \) are not rounded to the same number, we have:

\[
\text{uint64_t maxAbsErr} = 3 + \text{abs}(xe) + (\text{HI}(zpzpart) \gg (\text{POLYNOMIAL_PREC + IMPLICIT_ZEROS} + 11 - 64));
\]

\[
\text{uint64_t maxRelErr} = (\text{maxAbsErr} \gg (64 - u)) + 1;
\]

if \((((\text{mantissa} + \text{maxRelErr}) \wedge (\text{mantissa} - \text{maxRelErr})) \gg 11)\) {
    return log_rn_accurate (cstpart, dz, xe, argReduc1[ri].log_lo, argReduc2[si].log_lo);
}

/* Assemble the computed result */

\[
\text{uint64_t resultbits} = ((\text{uint64_t})\text{sign} \ll 63) + ((\text{uint64_t})(\text{exponent} + 1023) \ll 52) + (\text{mantissa} \gg 12) + ((\text{mantissa} \gg 11) \& 1); /* round to nearest */
\]

return (union { uint64_t u; double d; }){ resultbits }.d;
Second step

- Use 3 words instead of 2 for the precomputed log
- Use a much more accurate polynomial:
  - with coefficients on 128 bits instead of 64
    (but \(z\) is still only a 64-bit number)
  - and using a higher degree polynomial
Introduction and context

The Table Maker’s dilemma

One algorithm, many variants

Results

Bonus: a floating-point in, fixed-point out variant

Conclusions
A few Pareto points in the design space

<table>
<thead>
<tr>
<th>Table size (bytes)</th>
<th>degree 1st</th>
<th>degree 2nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>39,936</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>12,288</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4,032</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>2,240</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>2,016</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>900</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>594</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>298</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>glibc</td>
<td>crlibm-td</td>
</tr>
<tr>
<td>------------------</td>
<td>--------</td>
<td>-----------</td>
</tr>
<tr>
<td>degree pol. 1</td>
<td>3/8</td>
<td>6</td>
</tr>
<tr>
<td>degree pol. 2</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>tables size</td>
<td>13 Kb</td>
<td>8192 bytes</td>
</tr>
<tr>
<td>% accurate phase</td>
<td>N/A</td>
<td>1.5</td>
</tr>
</tbody>
</table>
### Average and max running time (in processor cycles)

#### Pentium timing

<table>
<thead>
<tr>
<th>cycles</th>
<th>MKL</th>
<th>glibc</th>
<th>crlibm</th>
<th>cr-de</th>
<th>cr-FixP</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg time</td>
<td>25</td>
<td>90</td>
<td>69</td>
<td>46</td>
<td>49</td>
</tr>
<tr>
<td>max time</td>
<td>25</td>
<td>11,554</td>
<td>642</td>
<td>410</td>
<td>79</td>
</tr>
</tbody>
</table>

#### Timing breakdown on two processors

<table>
<thead>
<tr>
<th>cycles</th>
<th>Core i5</th>
<th>Bostan</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>glibc</td>
<td>newlib</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>105</td>
</tr>
<tr>
<td>quick phase alone</td>
<td>42</td>
<td>94</td>
</tr>
<tr>
<td>accurate phase alone</td>
<td>74</td>
<td>181</td>
</tr>
<tr>
<td>both phases (avg time)</td>
<td>49</td>
<td>121</td>
</tr>
<tr>
<td>both phases (max time)</td>
<td>79</td>
<td>225</td>
</tr>
</tbody>
</table>

Slanted means: no correct rounding
Conclusion of this experiment

- Improvement in the range reduction thanks to a wider format
- ... leading to improvements in polynomial degree and table size
- Improvement in the rounding test
- Improvement in the worst-case evaluation time
- Probability to launch 2nd step is high,
  but this is acceptable since 2nd step is so cheap
- A branchless correctly rounded variant that is better than the glibc
Outline

Introduction and context

The Table Maker’s dilemma

One algorithm, many variants

Results

Bonus: a floating-point in, fixed-point out variant

Conclusions
**Motivation**

TKF91: DNA sequence alignment algorithm
- dynamic programming algorithm: alignment as a path within a 2D array.
- borders of an array initialized with log-likelihoods
- then array filled using recurrence formulae that involve only max and + operations.

All current implementations of this algorithm use a floating-point array, but
- `int64 +` and `max` are 1-cycle, vectorizable, and exact operations;
- absolute accuracy of initialization logs: up to $2^{-42}$ with FP log, $2^{-52}$ with FixP log.
Floating-point in, fixed-point out

- **output**: fixed-point, 12 bits integer part, 52 bit fractional part

  ![Fixed-point format diagram]

  - integer part
  - fraction

- **faithful**: target absolute accuracy $2^{-52}$

<table>
<thead>
<tr>
<th>output format</th>
<th>absolute accuracy</th>
<th>table size</th>
<th>Core i5 cycles</th>
<th>Bostan cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fix64</td>
<td>$2^{-52}$</td>
<td>2304</td>
<td>24</td>
<td>66</td>
</tr>
<tr>
<td>Fix128</td>
<td>$2^{-116}$</td>
<td>4032</td>
<td>60</td>
<td>179</td>
</tr>
<tr>
<td>double (libm)</td>
<td>$2^{-42}$</td>
<td>90</td>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

- Fix64 is the code of the first step only, without the conversion to float.
- **tweak**: poly degree 3 only for abs. accuracy $2^{-59}$
- Fix128 is the code of the second step only, without the conversion to float.
Only partial experiments

- Improvement in accuracy measured
- No noticeable improvement in performance
Outline

Introduction and context

The Table Maker’s dilemma

One algorithm, many variants

Results

Bonus: a floating-point in, fixed-point out variant

Conclusions
Conclusion

- Competitive against state-of-the-art
- 2nd step faster than other implementations
- Possible to do only the second step
- Better argument reduction

Limitations:
Conclusion

- Competitive against state-of-the-art
- 2nd step faster than other implementations
- Possible to do only the second step
- Better argument reduction

Limitations:
- Less portable than floating-point
- No support for vectorization
Conclusion

- Competitive against state-of-the-art
- 2nd step faster than other implementations
- Possible to do only the second step
- Better argument reduction

Limitations:
- Less portable than floating-point
- No support for vectorization
- Minimize latency, not throughput
Future work

Going further with the logarithm

- Computing the worst-cases for absolute precision
- Finishing the Gappa proof (solution reconstruction)
- Trying variant without the cancellations
- Implementing the log in *Metalibm*
- Comparing with the log already in *Metalibm*, or on other platforms

J. Le Maire, F. de Dinechin and J.-M. Muller

Computing correctly rounded logarithm with fixed-point operations
Future work

Going further with the logarithm

- Computing the worst-cases for absolute precision
- Finishing the Gappa proof (solution reconstruction)
- Trying variant without the cancellations
- Implementing the log in *Metalibm*
- Comparing with the log already in *Metalibm*, or on other platforms

Going further with the fixed-point arithmetic

- Having a log returning a fixed-point result (be it on two words)
Future work

Going further with the logarithm

- Computing the worst-cases for absolute precision
- Finishing the Gappa proof (solution reconstruction)
- Trying variant without the cancellations
- Implementing the log in Metalibm
- Comparing with the log already in Metalibm, or on other platforms

Going further with the fixed-point arithmetic

- Having a log returning a fixed-point result (be it on two words)
- Implementing other functions with fixed-point (sinpi, cospi)
Thanks for your attention

Any question?
Reconstructing the solution

\[ e \cdot \ln(2) : \]

\[ \ln(\text{inv}_x^{-1}) : \]

\[ \ln(\text{inv}_y^{-1}) : \]

\[ P(z) \approx \ln(1 + z) : \]

sum:
Reconstructing the solution

\[ e \cdot \ln(2): \]

\[ \ln(\text{inv}_x^{-1}): \]

\[ \ln(\text{inv}_y^{-1}): \]

\[ P(z) \approx \ln(1 + z): \]

\[ \text{sum:} \]

\[ \frac{1}{11} \text{ floating-point fraction} \]
Reconstructing the solution

\[ e \cdot \ln(2): \]

\[ \ln(\text{inv}_x^{-1}): \]

\[ \ln(\text{inv}_y^{-1}): \]

\[ P(z) \approx \ln(1 + z): \]

\[ \text{sum:} \]

1 floating-point fraction

J. Le Maire, F. de Dinechin and J.-M. Muller
Computing correctly rounded logarithm with fixed-point operations
Reconstructing the solution

\[ e \cdot \ln(2) : \]

\[ \ln(\text{inv}_x^{-1}) : \]

\[ \ln(\text{inv}_y^{-1}) : \]

\[ P(z) \approx \ln(1 + z) : \]

\[ \text{sum:} \]

1 floating-point fraction
Reconstructing the solution

\[ e \cdot \ln(2): \]

\[ \ln(\text{inv}_x^{-1}): \]

\[ \ln(\text{inv}_y^{-1}): \]

\[ P(z) \approx \ln(1 + z): \]

sum:

11 0 -11 -53 -117

-1 floating-point fraction
Reconstructing the solution

\[ e \cdot \ln(2): \]

\[ \ln(\text{inv}_x^{-1}): \]

\[ \ln(\text{inv}_y^{-1}): \]

\[ P(z) \approx \ln(1 + z): \]

\[ \text{sum:} \]

\[ 1 \text{ floating-point fraction} \]
Example of code 2

```c
/* X = 2^xe * (xbits/2^52) */
x_e = 1023;
x_bits = (x_bits & 0xFFFFFFFFFFFFFFFFFull) + (UINT64_C(1) << 52);

/* X = 2^xe * (1/R) * Y, 
with Y = y/2^(52 + ARG_REDUCE_1_SIZE) 
and 1/R = argReduc1[ri].val/2^ARG_REDUCE_1_SIZE */
uint8_t ri = (x_bits >> (52 - ARG_REDUCE_1_PREC)) - (1u << ARG_REDUCE_1_PREC);
uint64_t y = ARG_REDUCE_1_GETVALUE(ri) * x_bits;

/* Y = (1/S) * (1 + dZ), 
with dZ = dz/2^(52 + ARG_REDUCE_1_SIZE + ARG_REDUCE_2_SIZE) 
and 1/S = argReduc2[si].val/2^ARG_REDUCE_2_SIZE */
uint8_t si = (y >> (52 + ARG_REDUCE_1_SIZE - ARG_REDUCE_2_PREC)) - (1u << ARG_REDUCE_2_PREC);
uint64_t dz = ARG_REDUCE_2_GETVALUE(si) * y; // the integer part of the fixed-point is removed by overflow

/* Compute part of the result that don't depend on Z (xe*log(2) + log(1/Ri) + log(1/Si)) */
uint128_t cstpart = fullimul(xe, log2fw_mid) + UINT128((int64_t)xe*log2fw_high, 0) // don't need a full mul here 
+ UINT128(argReduc1[ri].log_hi, argReduc1[ri].log_mid) 
+ UINT128(argReduc2[si].log_hi, argReduc2[si].log_mid);

/* Polynomial approximation of log(1+Z)/Z ~ P(Z), and evaluate Z*P(Z) */
uint64_t p = UINT64_C(0xfffffffffffffff4) 
-(highmul(dz, 
  UINT64_C(0x7fffffff091895)
  -(highmul(dz, 
    UINT64_C(0x55555509230fb34c)
     -(highmul(dz,UINT64_C(0x3ff8f2ad563f0e19)>>IMPLICIT_ZEROS) 
    )>>IMPLICIT_ZEROS))>>IMPLICIT_ZEROS);
uint128_t zpzpart = fullmul(dz, p);
```

J. Le Maire, F. de Dinechin and J.-M. Muller

Computing correctly rounded logarithm with fixed-point operations

48
Assemble the two parts, compute the sign, mantissa and exponent

```c
uint128_t longres = cstpart + (zpzpart >> (11 + IMPLICIT_ZEROS));
uint64_t sign = -(HI(longres) >> 63); // sign is 0 if result > 0, and ~0 otherwise
// if sign != 0, this is longres = ~longres: it approximate the absolute value (~a = ~a + 1)
// to avoid the approximation, do: longres = ((uint64_t)sign + longres) ^UINT128(sign, sign);
longres ^= UINT128(sign, sign);

int u = clz64(HI(longres)) + 1;
int exponent = 11 - u;
uint64_t mantissa = HI(longres << u);
```

Compute the maximal absolute error (aligned with longres)

```c
uint64_t maxAbsErr = 3 + abs(xe) + (HI(zpzpart) >> (POLYNOMIAL_PREC + IMPLICIT_ZEROS + 11 - 64));
uint64_t maxRelErr = (maxAbsErr >> (64 - u)) + 1;
if (((mantissa + maxRelErr) ^ (mantissa - maxRelErr)) >> 11) {
    return log_rn_accurate(cstpart, dz, xe, argReduc1[ri].log_lo, argReduc2[si].log_lo);
}
```

Assemble the computed result

```c
uint64_t resultbits = ((uint64_t)sign << 63) + ((uint64_t)(exponent+1023) << 52) + (mantissa >> 12) + ((mantissa >> 11) & 1); /* round to nearest */
return (union { uint64_t u; double d; }){ resultbits }.d;
```