

Computing correctly rounded logarithms with fixed-point operations

Julien Le Maire, Florent de Dinechin,
Jean-Michel Muller and Nicolas Brunie



Outline

Introduction and context

Algorithm

Results and comparisons for libm `log`

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

Conclusions

Software evaluating elementary function

This work is about `libm` functions:

- **prototype:**

floating-point in, floating-point out, e.g.

```
double log(double x);
```

Software evaluating elementary function

This work is about `libm` functions:

- **prototype:**

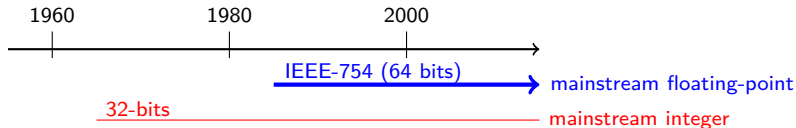
floating-point in, floating-point out, e.g.

```
double log(double x);
```

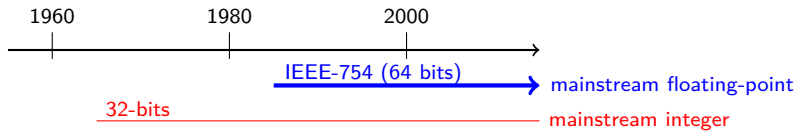
- **implementation:**

Nearly 100% of the implementations and literature use floating-point

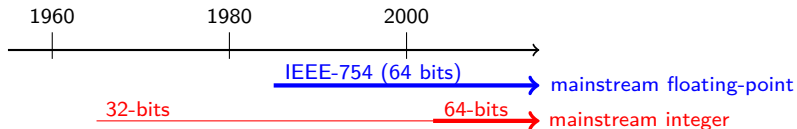
- Integer-based implementations: only on processors without FPU (StrongArm, ST200)



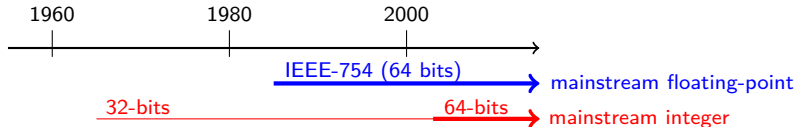
The times they are a-changing



The times they are a-changing

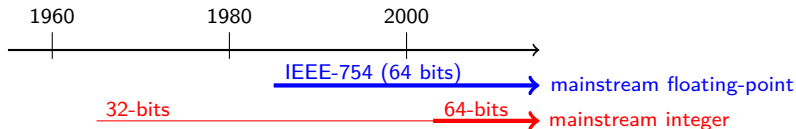


The times they are a-changing



At the same time, architectural changes such as
unified integer/floating-point registers.

The times they are a-changing



At the same time, architectural changes such as
unified integer/floating-point registers.

This work:

Re-evaluate the idea of implementing floating-point functions using integer arithmetic.

Integer now seems better than floating-point

- most operations are faster on integers, especially **addition**
 - 3-5 cycles in floating point
 - 1 cycle in integer (more or less defines the processor cycle time)

Integer now seems better than floating-point

- most operations are faster on integers, especially **addition**
 - 3-5 cycles in floating point
 - 1 cycle in integer (more or less defines the processor cycle time)
- 64 bits of significand is better than 52
 - if you can predict the value of the exponent, exponent bits are wasted bits
 - convert precision to speed?

Integer now seems better than floating-point

- most operations are faster on integers, especially **addition**
 - 3-5 cycles in floating point
 - 1 cycle in integer (more or less defines the processor cycle time)
- 64 bits of significand is better than 52
 - if you can predict the value of the exponent, exponent bits are wasted bits
 - convert precision to speed?
- modern 64-bit machines offer all the integer instructions we need
 - addition
 - multiplication $64 \times 64 \rightarrow 128$ (**mulq**)
 - count leading zeroes, shifts (**lzcnt**, **bsr**)

Integer now seems better than floating-point

- most operations are faster on integers, especially **addition**
 - 3-5 cycles in floating point
 - 1 cycle in integer (more or less defines the processor cycle time)
- 64 bits of significand is better than 52
 - if you can predict the value of the exponent, exponent bits are wasted bits
 - convert precision to speed?
- modern 64-bit machines offer all the integer instructions we need
 - addition
 - multiplication $64 \times 64 \rightarrow 128$ (**mulq**)
 - count leading zeroes, shifts (**lzcnt**, **bsr**)
- Fast small multiprecision out of the box:
 - mainstream compilers (**gcc**, **clang**, **icc**) support `__int_128`
 - addition $128 \times 128 \rightarrow 128$ (**add**, **adc**)
 - shift on two registers (**shld**, **shrd**)
 - multiplication, etc...

No vectorization yet

Integer SIMD/vector support still lagging behind FP

- until recently, no vector multiplication
- AVX512: 52-bit vector multiplication

(recycling mantissa multiplier)

So all hope is not left.

This work: an experiment

Implementing the *floating-point* logarithm function

- using only *integer* arithmetic
- for *performance*

(previous work motivated by *lack of FP hardware*)

- with state of the art *accuracy*: correct rounding

This work: an experiment

Implementing the *floating-point* logarithm function

- using only *integer* arithmetic
- for *performance*

(previous work motivated by *lack of FP hardware*)

- with state of the art *accuracy*: correct rounding

Why the log? Because it seemed the easiest function for this.

Main results

- worst-case execution time now in a **factor 3** of the best faithful implementations
 - An improvement of a factor 5 over previous state of the art
- average time **almost twice better** than current glibc.
- proposal of a **floating-point in, fix-point out** variant of the log function

Outline

Introduction and context

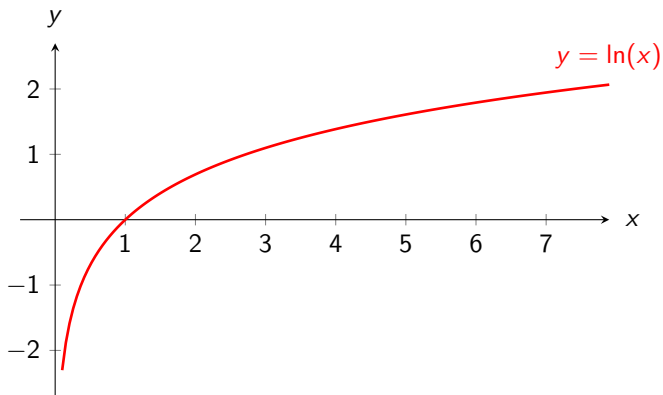
Algorithm

Results and comparisons for libm `log`

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

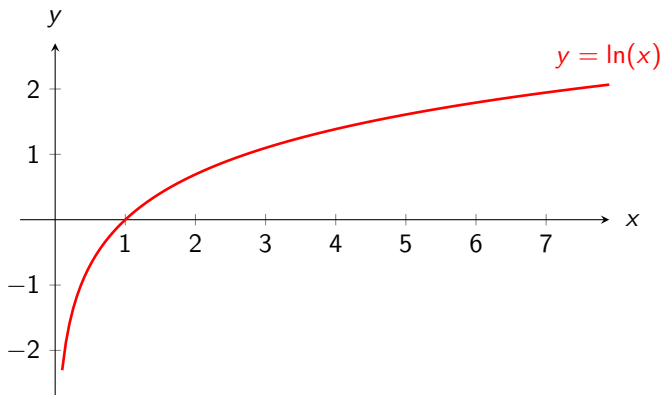
Conclusions

Logarithm, the mathematical version



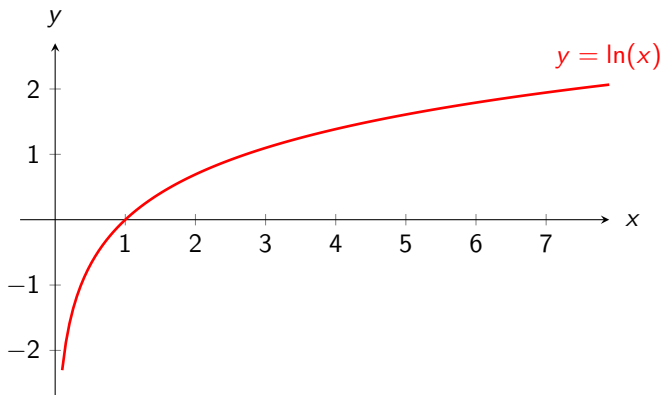
Logarithm, the mathematical version

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



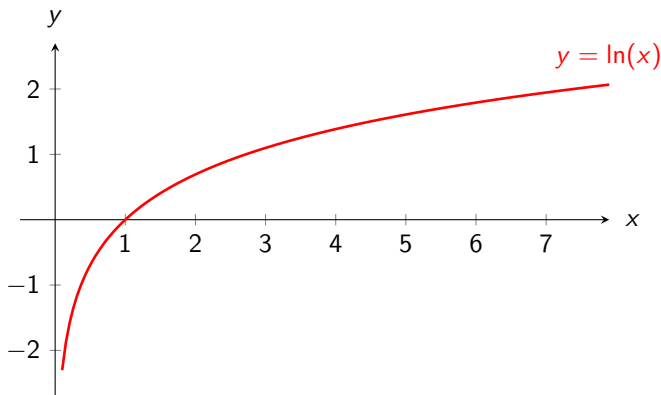
Logarithm, the mathematical version

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



Logarithm, the mathematical version

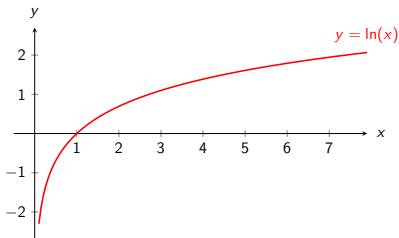
- $\ln(a \times b) = \ln(a) + \ln(b)$
- $\ln(b^a) = a \times \ln(b)$
- Taylor: for x small, $\ln(1 + x) \approx x - x^2/2 + x^3/3...$



Logarithm, the floating-point version

The floating point version of the natural logarithm is called \log
(you will also find \log_2 and \log_{10} and a few others)

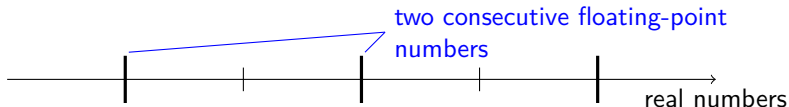
$$\forall x \in \mathbb{F}_{64} \quad \log(x) = \circ(\ln(x))$$



On-demand accuracy

Muller and Lefèvre solved the table maker dilemma for \log

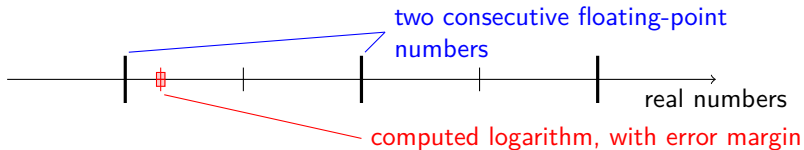
Computing the log with an error $\leq 2^{-113}$ enables correct rounding



On-demand accuracy

Muller and Lefèvre solved the table maker dilemma for \log

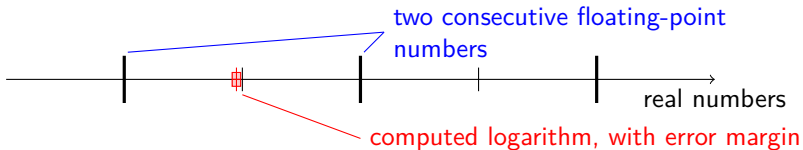
Computing the log with an error $\leq 2^{-113}$ enables correct rounding



On-demand accuracy

Muller and Lefèvre solved the table maker dilemma for \log

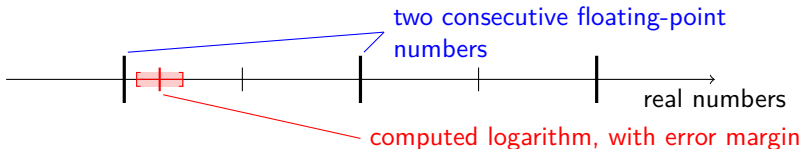
Computing the log with an error $\leq 2^{-113}$ enables correct rounding



On-demand accuracy

Muller and Lefèvre solved the table maker dilemma for \log

Computing the log with an error $\leq 2^{-113}$ enables correct rounding



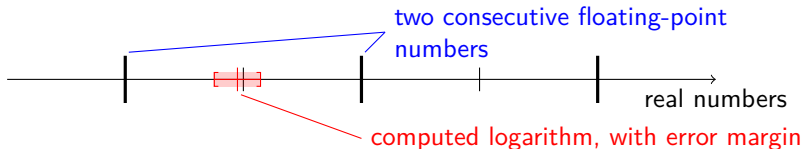
CRLibm refinement of Ziv's technique:

- First step: quick-and-dirty evaluation of $\ln(x)$
(just accurate enough to ensure correct rounding in most cases)

On-demand accuracy

Muller and Lefèvre solved the table maker dilemma for \log

Computing the log with an error $\leq 2^{-113}$ enables correct rounding



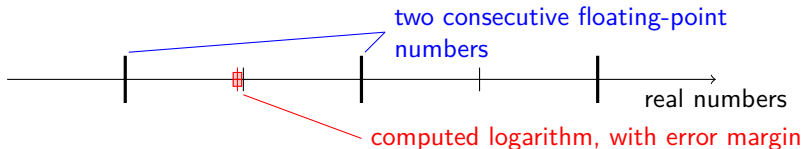
CRLibm refinement of Ziv's technique:

- 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

On-demand accuracy

Muller and Lefèvre solved the table maker dilemma for \log

Computing the log with an error $\leq 2^{-113}$ enables correct rounding



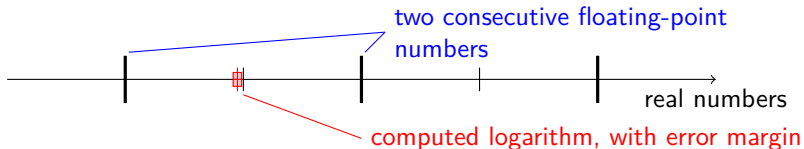
CRLibm refinement of Ziv's technique:

- 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

On-demand accuracy

Muller and Lefèvre solved the table maker dilemma for \log

Computing the log with an error $\leq 2^{-113}$ enables correct rounding



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

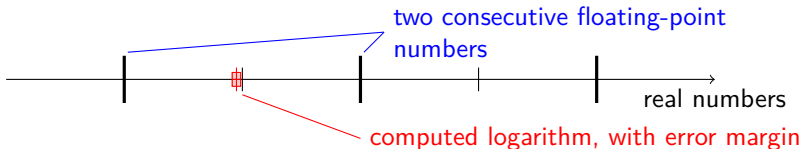
Trade-off between first and second steps:

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

On-demand accuracy

Muller and Lefèvre solved the table maker dilemma for \log

Computing the log with an error $\leq 2^{-113}$ enables correct rounding



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:

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

Best so far: $\text{Time}(2\text{nd step}) \approx 10 \times \text{Time}(1\text{st step})$

This work: $\text{Time}(2\text{nd step}) \approx 2 \times \text{Time}(1\text{st step})$

The big picture

1. Filter special cases (negative numbers, ∞ , ...)
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.

First argument range reduction

$$\begin{aligned} \text{input} &= 2^E \cdot (1 + x) \\ \ln(\text{input}) &= E \cdot \ln(2) + \ln(1 + x) \end{aligned}$$

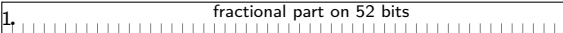
First argument range reduction

$$\begin{aligned} \text{input} &= 2^E \cdot (1 + x) \\ \ln(\text{input}) &= E \cdot \ln(2) + \ln(1 + x) \end{aligned}$$

Evaluation algorithm:

- approximate $\ln(1 + x)$ with a polynomial $p(x)$
degree needed: at least 26
- evaluate $E \cdot \ln(2)$
- add both terms

Tang's range reduction

$1 + x$:  fractional part on 52 bits

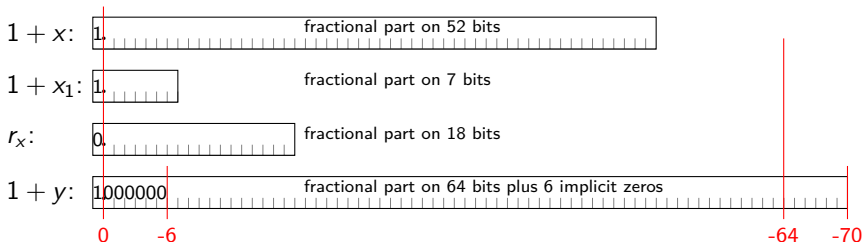
$1 + x_1$:  fractional part on 7 bits

r_x :  fractional part on 18 bits

- A table, addressed by x_1 , the 7 most significant bits of x , stores

$$r_x \approx \frac{1}{1 + x_1} \approx \frac{1}{1 + x} \quad \text{and} \quad \ln \left(\frac{1}{r_x} \right)$$

Tang's range reduction

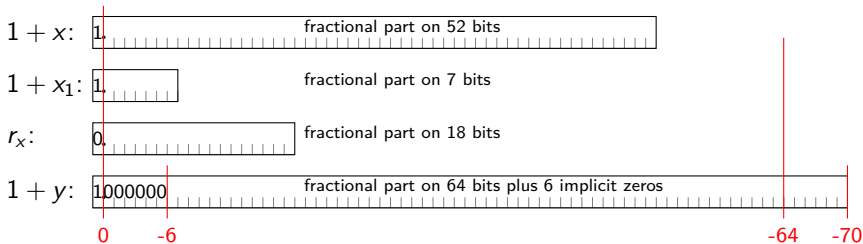


- A table, addressed by x_1 , the 7 most significant bits of x , stores

$$r_x \approx \frac{1}{1 + x_1} \approx \frac{1}{1 + x} \quad \text{and} \quad \ln\left(\frac{1}{r_x}\right)$$

- Define $1 + y = r_x \cdot (1 + x) \approx 1$
- Then $\ln(1 + x) = \ln(1 + y) + \ln\left(\frac{1}{r_x}\right)$

Tang's range reduction algorithm



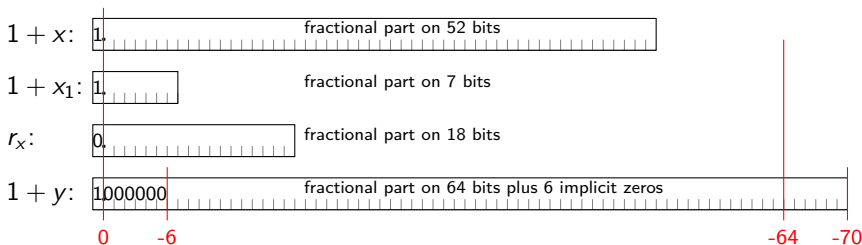
- Extract the index x_1
- Read, from a table addressed by x_1 , both r_x and $\ln(\frac{1}{r_x})$
- compute $y = r_x \cdot (1 + x) - 1$ (exactly)
- approximate $\ln(1 + y)$ with a polynomial $p(y)$

Degree needed: 8

- add it all:

$$\ln(\text{input}) \approx E \cdot \ln(2) + p(y) + \ln\left(\frac{1}{r_x}\right)$$

Tang's range reduction

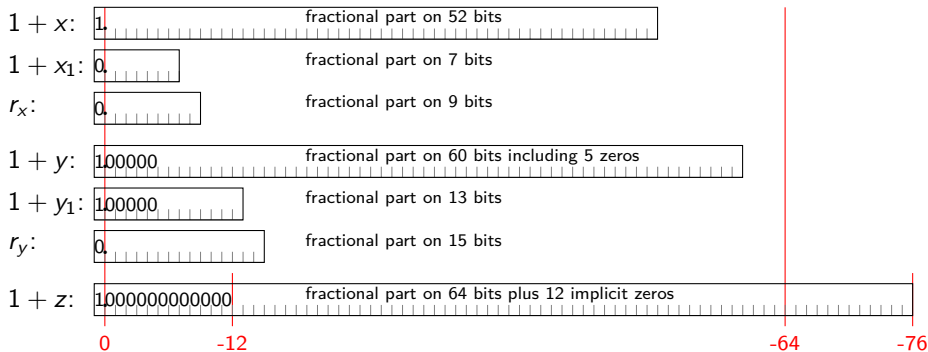


$$y = r_x \cdot (1 + x) - 1$$

With $1 + x$ on 53 bits we can tabulate r_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.

Two levels of Tang reduction



$$x \in [0, 1)$$

$$y \in [0, 2^{-6.41504})$$

$$z \in [0, 2^{-12.6747})$$

x_1 takes 64 different values

y_1 takes 96 different values

the whole reduction of x to z is **computed exactly** in 64-bit int.

A few Pareto points in the design space

Table size (bytes)	degree 1st	degree 2nd
39,936	3	5
12,288	3	6
4,032	4	7
2,240	4	8
2,016	4	9
900	5	10
594	6	12
298	7	14

Why stop at two levels of reduction?

Answer is: diminishing return.

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

	interval of x	degree needed
No reduction	$[-1/2, 1/2]$	29
1 level	$[-2^{-7}, 2^{-7}]$	8
2 levels	$[-2^{-12}, 2^{-12}]$	4
3 levels	$[-2^{-18}, 2^{-18}]$	3

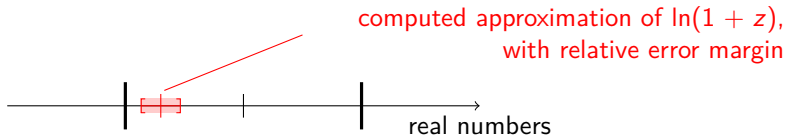
Adding more levels will cost more operations than it saves...

Polynomial approximation (advertisement)

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 \mathbb{C}

We obtain a Coq proof of the error:



Reconstructing the solution

$$\mathit{input} = 2^e \cdot (1 + x)$$

Reconstructing the solution

$$\mathit{input} = 2^e \cdot \frac{1}{r_x} \cdot (1 + y)$$

Reconstructing the solution

$$\text{input} = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z)$$

Reconstructing the solution

$$input = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z)$$

$$\ln(input) = e \cdot \ln(2) + \ln(r_x^{-1}) + \ln(r_y^{-1}) + \ln(1 + z)$$

Reconstructing the solution

$$\text{input} = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z)$$

$$\ln(\text{input}) = e \cdot \ln(2) + \ln(r_x^{-1}) + \ln(r_y^{-1}) + \ln(1 + z)$$

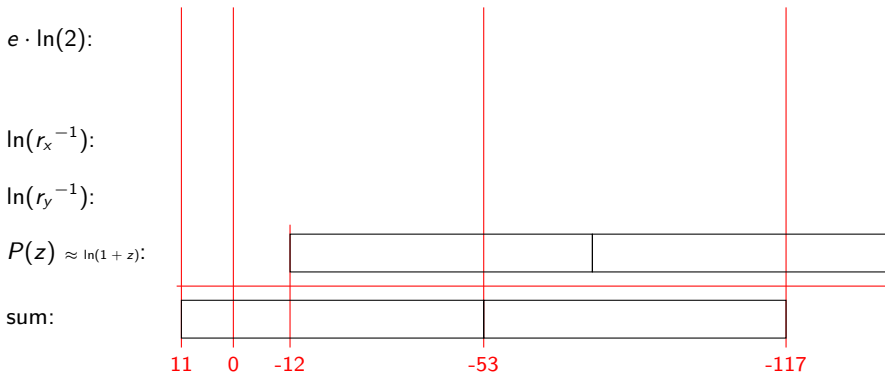


“If we can predict the exponents, exponent bits are wasted bits”

Reconstructing the solution

$$\text{input} = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z)$$

$$\ln(\text{input}) = e \cdot \ln(2) + \ln(r_x^{-1}) + \ln(r_y^{-1}) + \ln(1 + z)$$

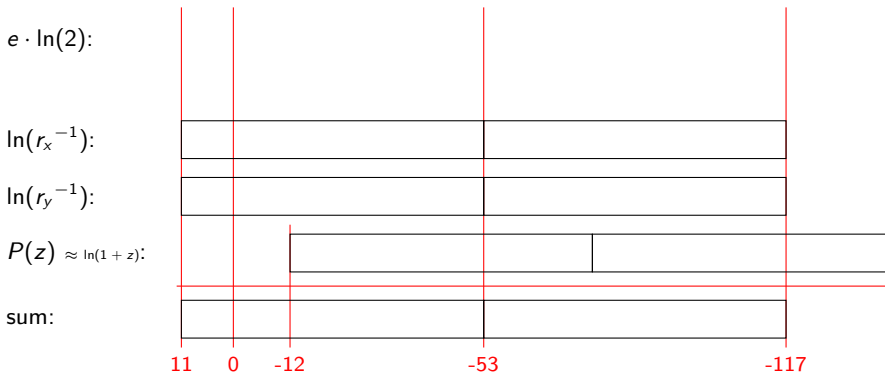


“If we can predict the exponents, exponent bits are wasted bits”

Reconstructing the solution

$$\text{input} = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z)$$

$$\ln(\text{input}) = e \cdot \ln(2) + \ln(r_x^{-1}) + \ln(r_y^{-1}) + \ln(1 + z)$$

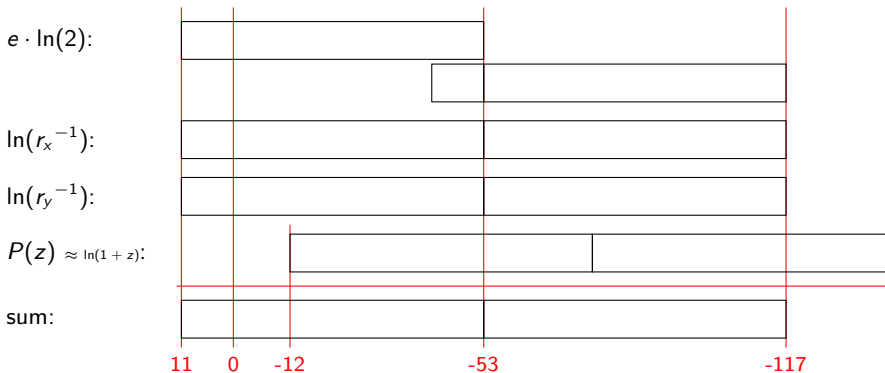


“If we can predict the exponents, exponent bits are wasted bits”

Reconstructing the solution

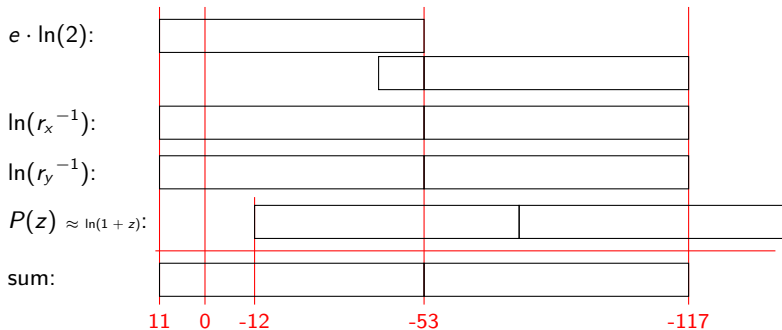
$$\text{input} = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z)$$

$$\ln(\text{input}) = e \cdot \ln(2) + \ln(r_x^{-1}) + \ln(r_y^{-1}) + \ln(1 + z)$$

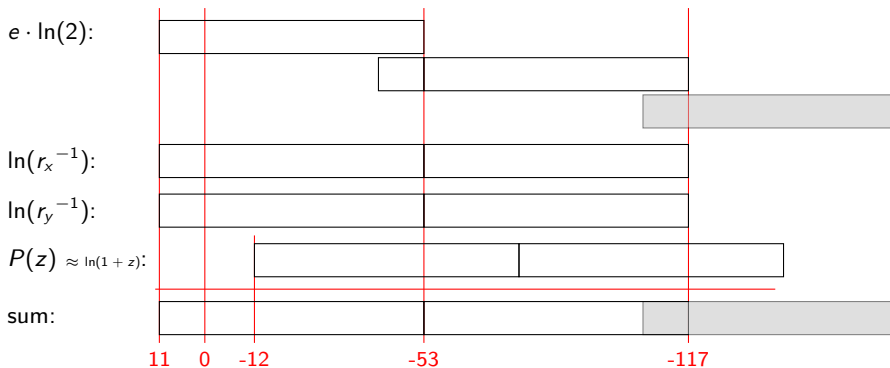


“If we can predict the exponents, exponent bits are wasted bits”

Error evaluation

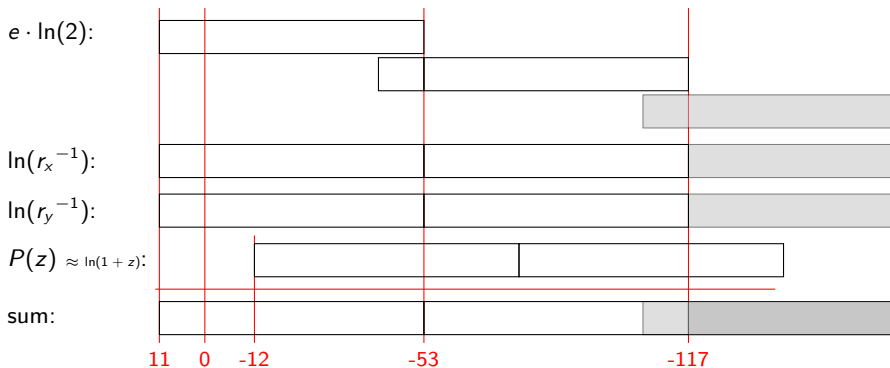


Error evaluation



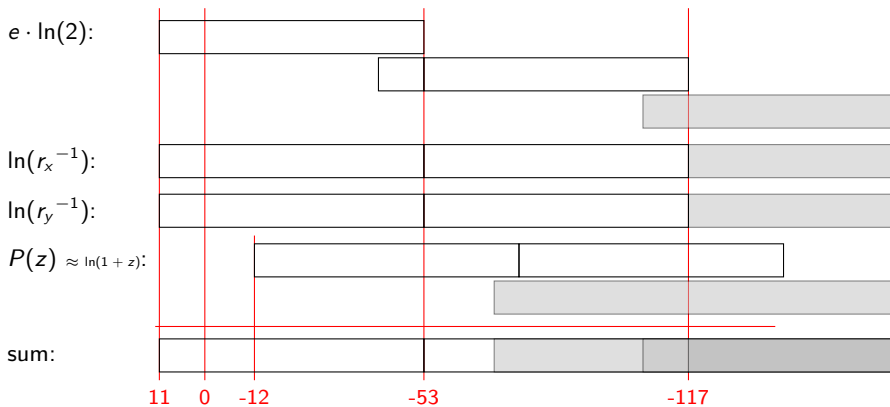
$$\epsilon < (|e|) \cdot 2^{-117}$$

Error evaluation



$$\epsilon < (|e| + 1 + 1) \cdot 2^{-117}$$

Error evaluation

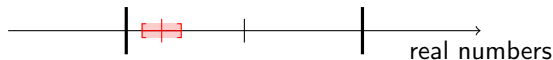


$$\epsilon < (|e| + 1 + 1 + P(z) \cdot 2^{-59}) \cdot 2^{-117}$$

Rounding test

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

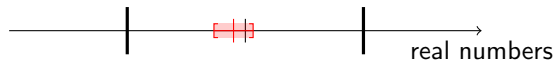
(two additions, a xor and a shift)



Rounding test

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

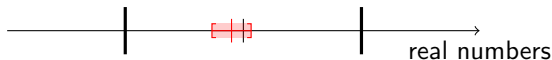
(two additions, a xor and a shift)



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)



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...

Second step

- Use 3 words instead of 2 for the precomputed $\ln(2)$
- 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 (7) polynomial

Outline

Introduction and context

Algorithm

Results and comparisons for libm `log`

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

Conclusions

Implementation parameters of correctly rounded implementations

	glibc	crlibm-td	crlibm-de	cr-FixP
degree pol. 1	3/8	6	7	4
degree pol. 2	20	12	14	7
tables size	13 Kb	8192 bytes	6144 bytes	4032 bytes
% accurate phase	N/A	1.5	0.4	4.4

Average and max runing time (in processor cycles)

Pentium timing (lower is better)

cycles	<i>MKL</i>	glibc	crlibm	cr-de	cr-FixP
avg time	25	90	69	46	49
max time	25	11,554	642	410	79

Timing breakdown on two processors

cycles	Core i5	Bostan
System	glibc 90	newlib 105
quick phase alone	42	94
accurate phase alone	74	181
both phases (avg time)	49	121
both phases (max time)	79	225

Slanted means: no correct rounding

Average and max runing time (in processor cycles)

Pentium timing (lower is better)

cycles	<i>MKL</i>	glibc	crlibm	cr-de	cr-FixP
avg time	25	90	69	46	49
max time	25	11,554	642	410	79

Timing breakdown on two processors

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

Slanted means: no correct rounding

Average and max runing time (in processor cycles)

Pentium timing (lower is better)

cycles	<i>MKL</i>	glibc	crlibm	cr-de	cr-FixP
avg time	25	90	69	46	49
max time	25	11,554	642	410	79

Timing breakdown on two processors

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

Slanted means: no correct rounding

Average and max runing time (in processor cycles)

Pentium timing (lower is better)

cycles	<i>MKL</i>	glibc	crlibm	cr-de	cr-FixP
avg time	25	90	69	46	49
max time	25	11,554	642	410	79

Timing breakdown on two processors

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

Slanted means: no correct rounding

Outline

Introduction and context

Algorithm

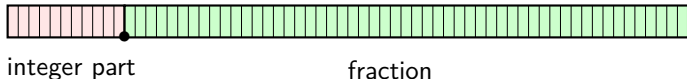
Results and comparisons for libm `log`

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

Conclusions

Floating-point in, fixed-point out

- output: fixed-point, 11 bits integer part, 53 bit fractional part



- target absolute accuracy 2^{-52}

output format	absolute accuracy	table size	Core i5 cycles	Bostan cycles
Fix64	2^{-52}	2304	24	66
Fix128	2^{-116}	4032	60	179
double (libm)	2^{-42}		90	105

- 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.

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.

Only partial experiments

- Improvement in accuracy measured
- No noticeable improvement in performance

Outline

Introduction and context

Algorithm

Results and comparisons for libm `log`

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

Conclusions

Conclusions

- Better range reduction thanks to a wider format
- ... leading to improvements in polynomial degree and table size
- Faster multiprecision due to higher precision
- Able to reuse some computations of the fast step in the accurate step
- Alternative rounding test for accurate step
- Probability to launch 2nd step is high,
but this is acceptable since 2nd step is so cheap

Conclusions

- Better range reduction thanks to a wider format
 - ... leading to improvements in polynomial degree and table size
 - Faster multiprecision due to higher precision
 - Able to reuse some computations of the fast step in the accurate step
 - Alternative rounding test for accurate step
 - Probability to launch 2nd step is high,
but this is acceptable since 2nd step is so cheap
-
- Competitive against state-of-the-art
 - Worst case improved compared to others implementations
 - Second step-only version is a viable alternative

Conclusions

- Better range reduction thanks to a wider format
 - ... leading to improvements in polynomial degree and table size
 - Faster multiprecision due to higher precision
 - Able to reuse some computations of the fast step in the accurate step
 - Alternative rounding test for accurate step
 - Probability to launch 2nd step is high,
but this is acceptable since 2nd step is so cheap
-
- Competitive against state-of-the-art
 - Worst case improved compared to others implementations
 - Second step-only version is a viable alternative

Limitations:

- Require 64 bits integer support
- No support for vectorization

Thanks for your attention

Any question ?

Outline

Some code

details on the solution reconstruction

Special cases: business as usual

```
/* reinterpret x to manipulate its bits more easily */
uint64_t inputbits = ((union { double d; uint64_t u; }){input}).u;
int xe = inputbits >> 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;
    }
}
xe -= 1023;
```

Only interesting line: the subnormal management

Argument reduction

```
/*  $input = 2^{xe} * (1 + X)$  */
uint64_t x = inputbits & 0xFFFFFFFFFFFFFFFF;

/*  $1 + X = (1/Ri) * Y$  */
uint8_t x1 = x >> (52 - ARG_REDUCE_1_PREC);
uint64_t y = argReduce1_inv[x1] * (x + (1ull << 52));
__builtin_prefetch(& argReduce1_log[x1], 0, 0);

/*  $Y = (1/S) * (1 + Z)$  */
/* with  $dZ = dz/2^{(52 + ARG_REDUCE_1_SIZE + ARG_REDUCE_2_SIZE)}$ 
   and  $1/S = argReduce2[si].val/2^{ARG_REDUCE_2_SIZE}$  */
uint8_t y1 = (y >> (52 + ARG_REDUCE_1_SIZE - ARG_REDUCE_2_PREC))
             - (1u << ARG_REDUCE_2_PREC);
uint64_t z = argReduce2_inv[y1] * y; // +1 part removed by overflow
__builtin_prefetch(& argReduce2_log[y1], 0, 0);
```

Polynomial evaluation

```
/* Polynomial approximation of  $\log(1+Z)/Z \sim P(Z)$  and  $Z \cdot P(Z)$  */
static const uint64_t a4 = UINT64_C(0x3ffc147cb4539237);
static const uint64_t a3 = UINT64_C(0x5555553dc70f0dfd);
static const uint64_t a2 = UINT64_C(0x7fffffffdd574fd);
static const uint64_t a1 = UINT64_C(0xfffffffffffffa);
uint64_t pz = a1 - (highmul(z,
                        a2 - (highmul(z,
                                        a3 - (highmul(z, a4) >> 12)
                                    ) >> 12)
                    ) >> 12);
uint128_t zpz = fullmul(z, pz);
```

```
/* Polynomial approximation without shifts:
   replace  $\text{highmul}(z, a) \gg 12$  with  $\text{highmul}(z, a \gg 12)$  */
static const uint64_t a4 = UINT64_C(0x000000003ffc147);
static const uint64_t a3 = UINT64_C(0x000005555553dc6);
static const uint64_t a2 = UINT64_C(0x0007fffffffdd57);
static const uint64_t a1 = UINT64_C(0xfffffffffffffa);
uint64_t pz = a1 - highmul(z, a2 - highmul(z, a3 - highmul(z, a4)));
uint128_t zpz = fullmul(z, pz);
```

Reconstructing the solution

```
/* Compute part of the result that don't depend on Z
   (xe*log(2) + log(1/Ri) + log(1/Si)) */
uint128_t cstpart = fullmul(xe, log2fw_mid)
    + UINT128((int64_t)xe * log2fw_high, 0) // fullmul not needed
    + UINT128(argReduc1[ri].log_hi, argReduc1[ri].log_lo)
    + UINT128(argReduc2[si].log_hi, argReduc2[si].log_lo);

/* Polynomial approximation of log(1+Z)/Z ~ P(Z) and Z*P(Z) */
...

/* Assemble the two parts, compute sign, mantissa and exponent */
uint128_t longres = cstpart + (zpz >> (11 + IMPLICIT_ZEROS));
uint64_t sign = - (HI(longres) >> 63); // 0 or ~0
// if sign != 0, this is longres = ~ longres (-a = ~a + 1)
// to avoid the +1 approx, 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) | (LO(longres) >> (64 - u))
```

Rounding test and conversion

```
/* Compute the maximal absolute error (aligned with longres):
   + 2 + abs(xe)      for xe*log(2), log(1/Ri) and log(1/Si)
   + 1 + zpz >> (POLYNOMIAL_PREC+IMPLICIT_ZEROS+11) for the polyno
   If result*(1 + maxRelErr) are not rounded to the same number, we
uint64_t maxAbsErr = 3 + abs(xe) + (HI(zpz) >> (POLYNOMIAL_PREC +
uint64_t maxRelErr = (maxAbsErr >> (64 - u)) + 1;
if (((mantissa + maxRelErr) ^ (mantissa - maxRelErr)) >> 11) {
    return log_rn_accurate (cstpart, z, xe);
}

/* Assemble the computed result */
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;
```

Outline

Some code

details on the solution reconstruction

Reconstructing the solution

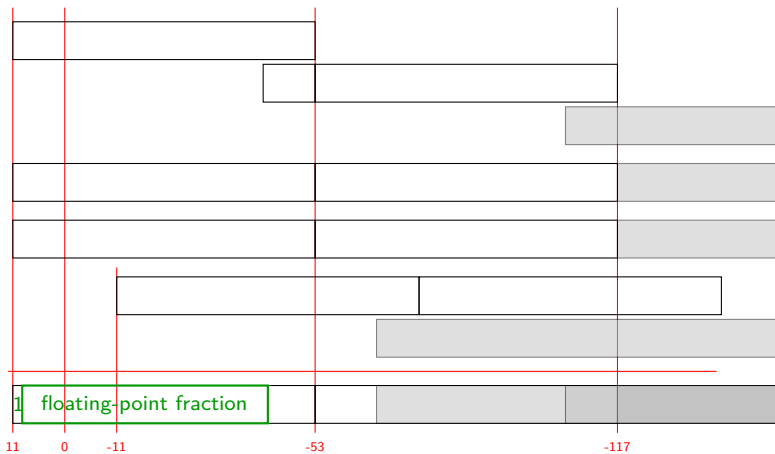
$e \cdot \ln(2)$:

$\ln(r_x^{-1})$:

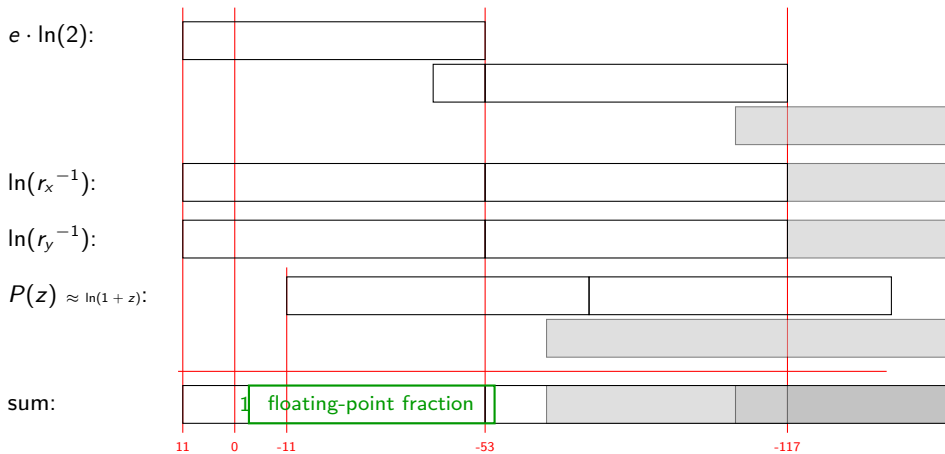
$\ln(r_y^{-1})$:

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

sum:



Reconstructing the solution



Reconstructing the solution

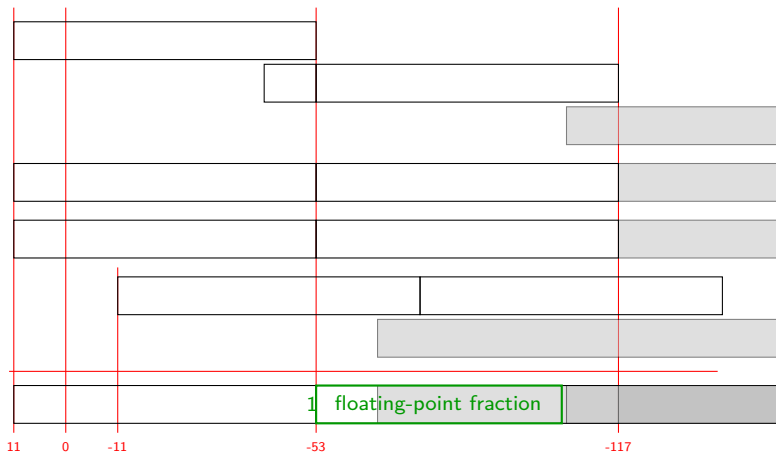
$e \cdot \ln(2)$:

$\ln(r_x^{-1})$:

$\ln(r_y^{-1})$:

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

sum:



Reconstructing the solution

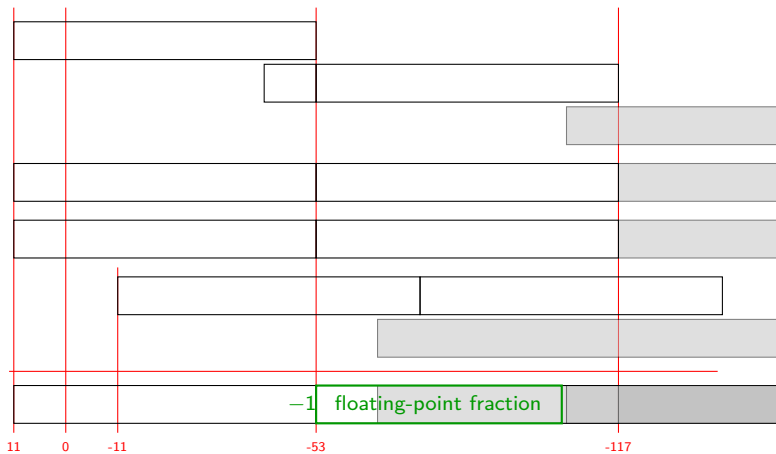
$e \cdot \ln(2)$:

$\ln(r_x^{-1})$:

$\ln(r_y^{-1})$:

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

sum:



Reconstructing the solution

$e \cdot \ln(2)$:

$\ln(r_x^{-1})$:

$\ln(r_y^{-1})$:

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

sum:

