Computing correctly rounded logarithms with fixed-point operations

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

Outline

Introduction and context

The Table Maker's dilemma

One algorithm, many variants

Results

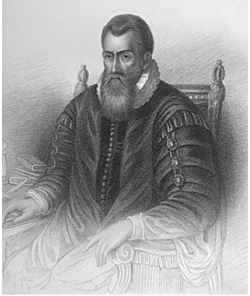
Bonus: a floating-point in, fixed-point out variant

Conclusions

Preparing 2017, international year of the logarithm

John Napier (aka Neper), 1550-1617

- popularized the use of the point in decimal notation



Preparing 2017, international year of the logarithm

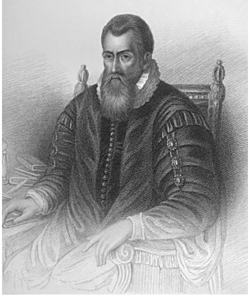
John Napier (aka Neper), 1550-1617

- popularized the use of the point in decimal notation
- *Mirifici Logarithmorum Canonis Descriptio* (1614)

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)



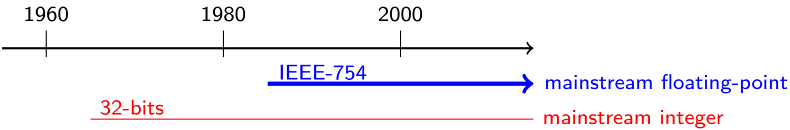
Celebrate a very specific year:

- 400th anniversary of Napier's death
- 6th logarithmic anniversary of the 1614 publication

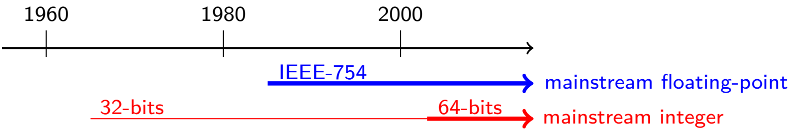
... with three amazing presentations this morning,

now doubt they will trigger many others.

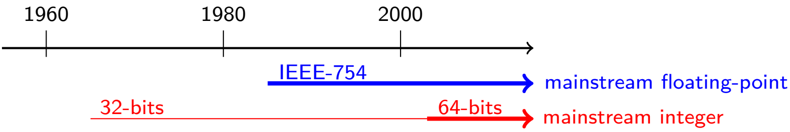
This talk is also about hardware and C



This talk is also about hardware and C



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*)

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.

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 $64x64 \rightarrow 128$ (*mulq*)
  - count leading zeroes, shifts (*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 $64x64 \rightarrow 128$ (*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?

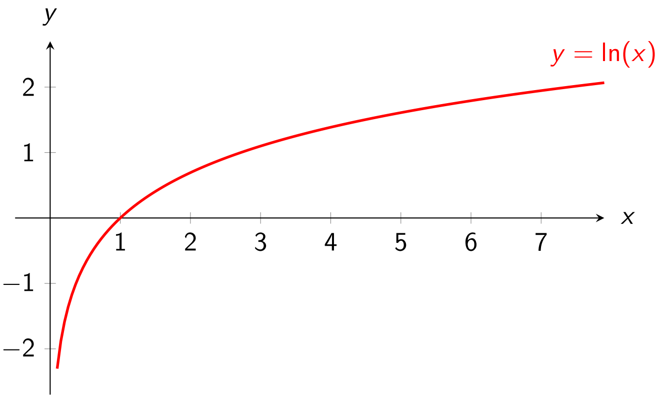
- 64-bit floating-point, but only 52-bit precision
  - if you can predict the value of the exponent, exponent bits are wasted bits.
- modern 64-bit machines offer all sort of useful integer instructions
  - addition
  - multiplication $64x64 \rightarrow 128$ (*mulq*)
  - count leading zeroes, shifts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

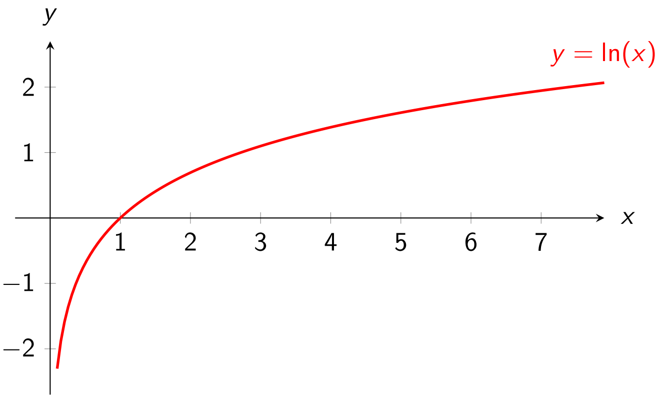
Caveat: integer SIMD/vector support still lagging behind FP
(no vector multiplication)

Logarithm, the mathematical version



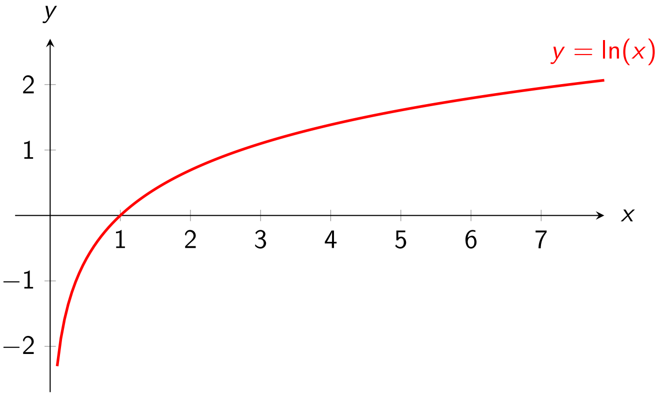
Logarithm, the mathematical version

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



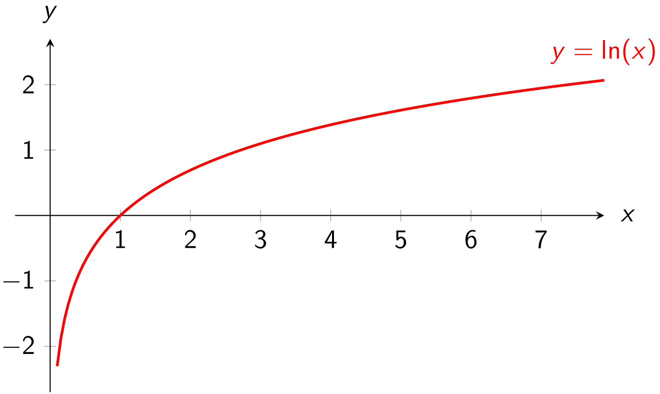
Logarithm, the mathematical version

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



Logarithm, the mathematical version

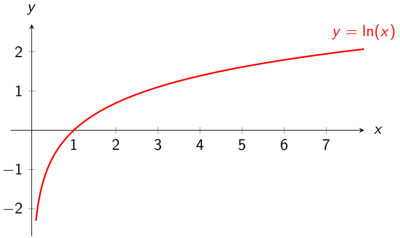
- $\ln(a \times b) = \ln(a) + \ln(b)$
- $\ln(b^a) = a \times \ln(b)$
- Taylor: for x small, $\ln(1+x) \approx x - x^2/2 + x^3/3 \ldots$



Logarithm, the floating-point version

The natural logarithm is called `log`

(you will also find `log2` and `log10` and a few others)

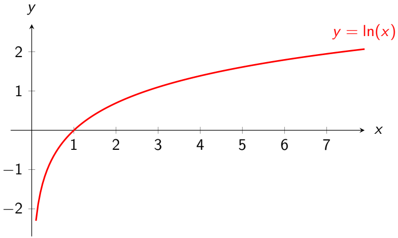


- Range: $\forall x \in \mathbb{F}_{64} \quad \texttt{log}(x) \in [-745, 710]$
  - looks like a waste of exponent bits...

Logarithm, the floating-point version

The natural logarithm is called `log`

(you will also find `log2` and `log10` and a few others)



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

Outline

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

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00003,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,02000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		160000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,38997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

The first digital signature algorithm

LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00003,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,02000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		160000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,38997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

- I want 12 significant digits

The first digital signature algorithm

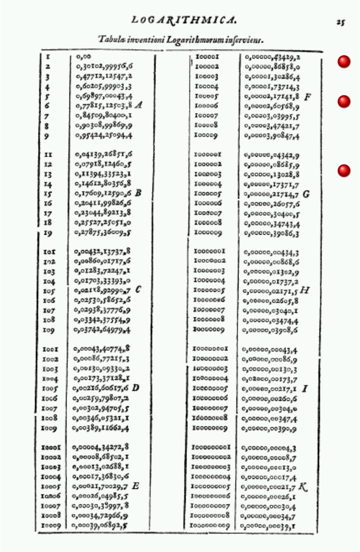
LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00003,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,02000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		160000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,39997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

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

The first digital signature algorithm



LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00	100001	0,00000,43429,2
2	0,30102,99956,6	100002	0,00000,86858,0
3	0,47712,12547,2	100003	0,00001,30286,4
4	0,60205,99903,3	100004	0,00001,73714,3
5	0,69897,00043,4	100005	0,00002,17141,8 *F*
6	0,77815,12503,8 *A*	100006	0,00002,60568,9
7	0,84509,80400,1	100007	0,00003,03995,5
8	0,90308,99869,9	100008	0,00003,47421,7
9	0,95424,25094,4	100009	0,00003,90847,4
11	0,04139,26851,6	1000001	0,00000,04342,9
12	0,07918,12460,5	1000002	0,00000,08685,9
13	0,11394,33523,1	1000003	0,00000,13028,8
14	0,14612,80356,8	1000004	0,00000,17371,7
15	0,17609,12590,6 *B*	1000005	0,00000,21714,7 *G*
16	0,20411,99826,6	1000006	0,00000,26057,6
17	0,23044,89213,8	1000007	0,00000,30400,5
18	0,25527,25051,0	1000008	0,00000,34743,4
19	0,27875,36009,5	1000009	0,00000,39086,3
101	0,00432,13737,8	10000001	0,00000,00434,3
102	0,00860,01717,6	10000002	0,00000,00868,6
103	0,01283,72247,1	10000003	0,00000,01302,9
104	0,01703,33393,0	10000004	0,00000,01737,2
105	0,02118,92990,7 *C*	10000005	0,00000,02171,5 *H*
106	0,02530,58652,6	10000006	0,00000,02605,8
107	0,02938,37776,9	10000007	0,00000,03040,1
108	0,03342,37554,9	10000008	0,00000,03474,4
109	0,03742,64979,4	10000009	0,00000,03908,6
1001	0,00043,40774,8	100000001	0,00000,00043,4
1002	0,00086,77215,3	100000002	0,00000,00086,9
1003	0,00130,09330,2	100000003	0,00000,00130,3
1004	0,00173,37128,1	100000004	0,02000,00173,7
1005	0,00216,60617,6 *D*	100000005	0,00000,00217,1 *I*
1006	0,00259,79807,2	100000006	0,00000,00260,6
1007	0,00302,94705,5	100000007	0,00000,00304,0
1008	0,00346,05321,1	100000008	0,00000,00347,4
1009	0,00389,11662,4	100000009	0,00000,00390,9
10001	0,00004,34272,8	1000000001	0,00000,00004,3
10002	0,00008,68502,1	1000000002	0,00000,00008,7
10003	0,00013,02688,1	1000000003	0,00000,00013,0
10004	0,00017,36830,6	1000000004	0,00000,00017,4
10005	0,00021,70029,7 *E*	1000000005	0,00000,00021,7 *K*
10006	0,00026,04985,5	1000000006	0,00000,00026,1
10007	0,00030,39997,8	1000000007	0,00000,00030,4
10008	0,00034,72966,9	1000000008	0,00000,00034,7
10009	0,00039,06892,5	1000000009	0,00000,00039,1

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

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

The first digital signature algorithm

LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00003,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,02000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		100000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,38997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

- 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

$$y = x, xxxxxxxxxxx17 \pm 10^{-14}$$

$$y = x, xxxxxxxxxxx83 \pm 10^{-14}$$

The first digital signature algorithm

LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00003,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,02000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		160000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,39997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

- 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

$$y = x, xxxxxxxxxxx17 \pm 10^{-14}$$

$$y = x, xxxxxxxxxxx83 \pm 10^{-14}$$

- Dilemma when

$$y = x, xxxxxxxxxxx50 \pm 10^{-14}$$

The first digital signature algorithm

LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00003,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,00000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		100000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,38997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

- 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

$$y = x, xxxxxxxxxxx17 \pm 10^{-14}$$

$$y = x, xxxxxxxxxxx83 \pm 10^{-14}$$

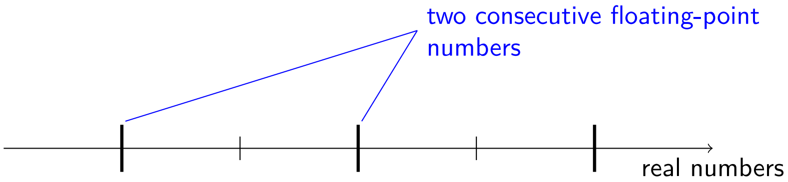
- Dilemma when

$$y = x, xxxxxxxxxxx50 \pm 10^{-14}$$

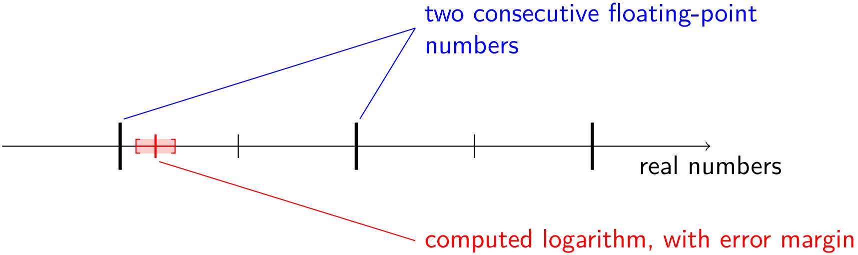
The first table-makers rounded these cases randomly,

and recorded them to confound copiers.

Solving the Table Maker's dilemma



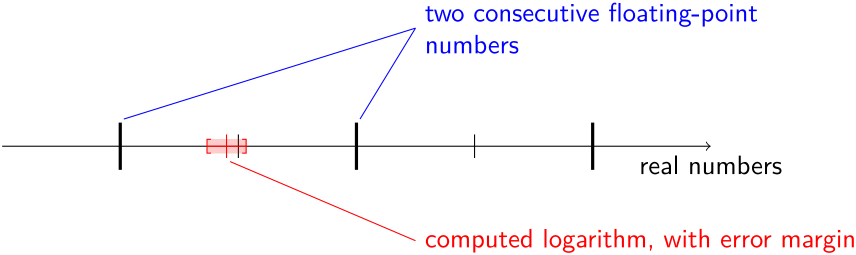
Solving the Table Maker's dilemma



$$y = x, xxxxxxxxxxx17 \pm 10^{-14}$$

Easy to round

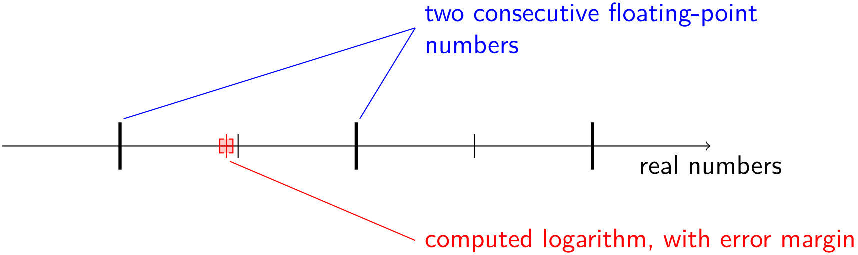
Solving the Table Maker's dilemma



$y = x, xxxxxxxxxxx50 \pm 10^{-14}$

Difficult to round

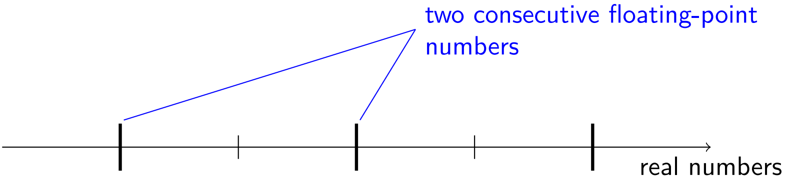
Solving the Table Maker's dilemma



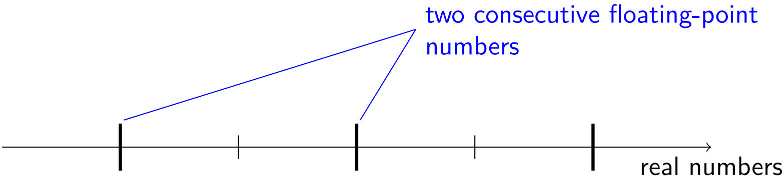
$y = x, xxxxxxxxxxx4996 \pm 10^{-16}$

Computing more accurately solves it

Solving the Table Maker's dilemma

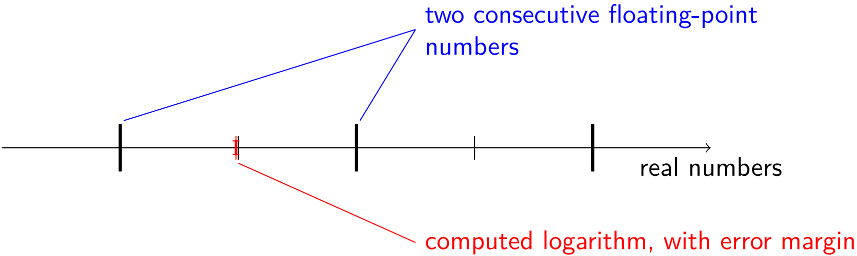


Solving the Table Maker's dilemma



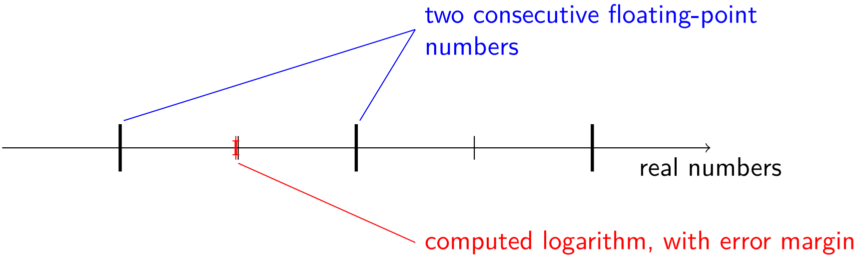
- $\forall x \in \mathbb{F}, \ln(x)$ is transcendental
- There is a finite number (2^{64}) of floating-point numbers.

Solving the Table Maker's dilemma



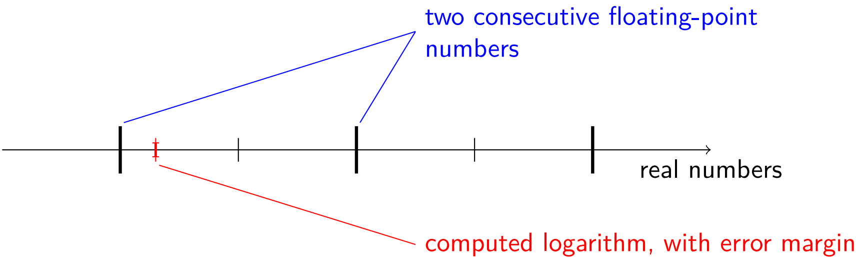
- $\forall x \in \mathbb{F}, \ln(x)$ is transcendental
- There is a finite number (2^{64}) of floating-point numbers.
- One of them is the worst to round

Solving the Table Maker's dilemma



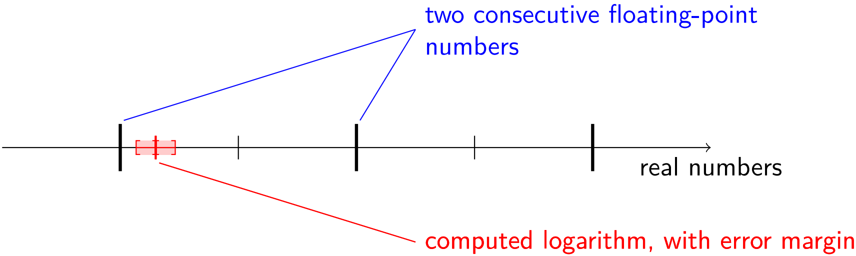
- $\forall x \in \mathbb{F}, \ln(x)$ is transcendental
- There is a finite number (2^{64}) of floating-point numbers.
- One of them is the worst to round
- Muller and Lefèvre computed that it requires an accuracy of 2^{-113}:
  evaluating the log to this accuracy enables correct rounding

Solving the Table Maker's dilemma



- $\forall x \in \mathbb{F}, \ln(x)$ 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
- but we don't need this accuracy for most cases
  (and it is more expensive to compute)

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
- but we don't need this accuracy for most cases
  (and it is more expensive to compute)

On-demand 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

On-demand 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:

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

On-demand 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:

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

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

Outline

Introduction and context

The Table Maker's dilemma

One algorithm, many variants

Results

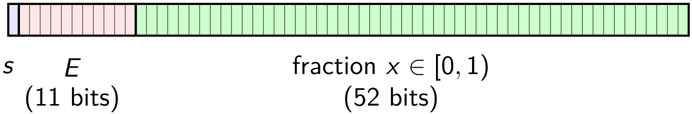
Bonus: a floating-point in, fixed-point out variant

Conclusions

The big picture

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

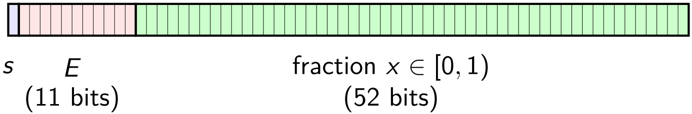
IEEE 754 floating-point



Value represented:

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

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

Special cases: businesss as usual

```c
/* reinterpret 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

$$input = 2^E \cdot (1 + x)$$
$$\ln(input) = E \cdot \ln(2) + \ln(1 + x)$$

First argument range reduction

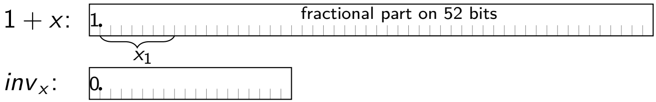
$$input = 2^E \cdot (1 + x)$$
$$\ln(input) = E \cdot \ln(2) + \ln(1 + x)$$

Evaluation algorithm:

- approximate $\ln(1 + x)$ with a polynomial $p(x)$

  degree needed: at least 26
- evaluate $E \cdot \ln(2)$
- add both terms

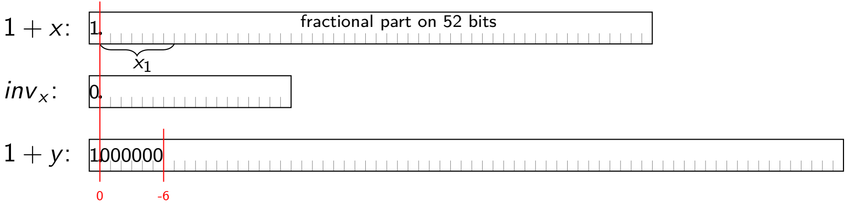
Tang's range reduction



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

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

Tang's range reduction



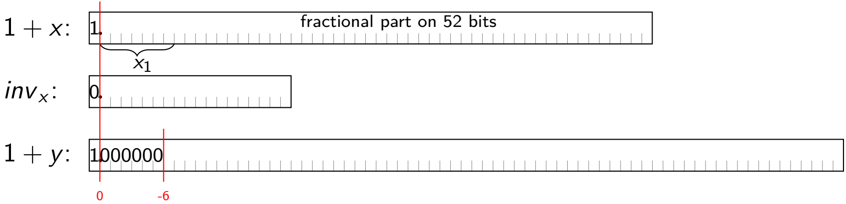
- A table, addressed by the x_1 most significand bits of x, stores

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

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

$$inv_x \cdot (1+x) \ = 1+y$$

Tang's range reduction



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

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

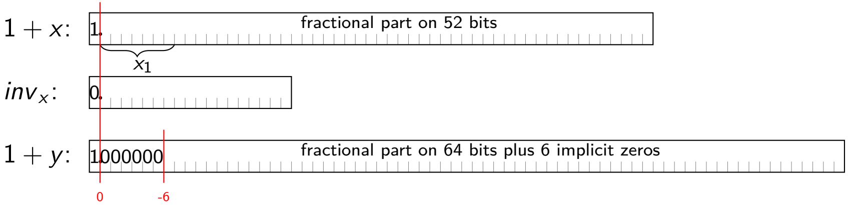
- As $inv_x \cdot (1+x) \approx 1$, define

$$inv_x \cdot (1+x) = 1+y$$

- Then

$$\ln(1+x) = \ln(1+y) - \ln(inv_x)$$

Tang's range reduction algorithm



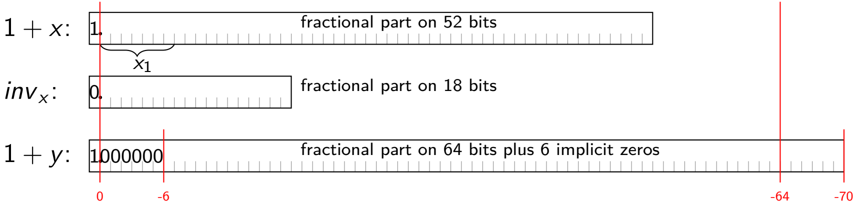
- Extract the index x_1
- Read, from a table addressed by x_1, both inv_x and $\ln(inv_x)$
- compute $y = inv_x \cdot (1 + x) - 1$ (exactly)
- approximate $\ln(1 + y)$ with a polynomial $p(y)$

Degree needed: 8

- add it all:

$$\ln(input) \approx E \cdot \ln(2) + p(y) - \ln(inv_x)$$

Here integers are better than floating-point



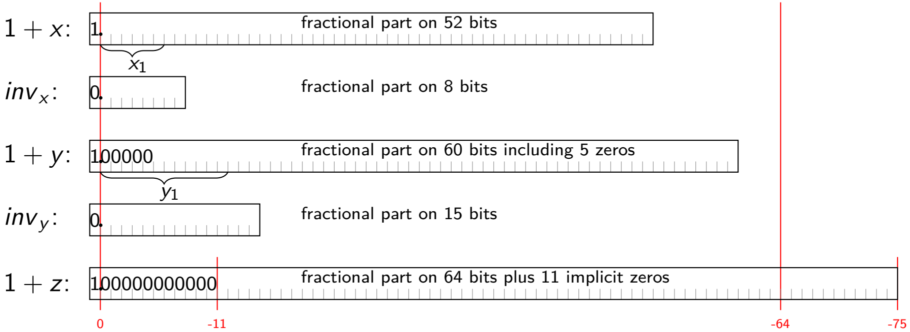
With a 53-bit $1+x$ we can tabulate inv_x on 18 bits:

- 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 $64 \times 64 \to 64$ multiplication.

Random remark about floating-point implementations of Tang's reduction

- There are reciprocal approximation instructions in most recent processors, including this pentium.
- Computing $y = inv_x \cdot (1 + x) - 1$ exactly requires an FMA, or double-extended, or a bit of double-FP arithmetic

Two levels of Tang reduction



$x \in [0, 1)$

$y \in \left[0, 2^{-5.41503}\right)$

$z \in \left[0, 2^{-11.8262}\right)$

x_1 *takes* 64 *different values*

y_1 *takes* 96 *different values*

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

Ugly code 2: 2 levels of Tang's reduction

```
/* X = 2^xe * (1/R) * Y,
 with  Y = y/2^(52 + ARG_REDUC_1_SIZE)
 and 1/R = argReduc1[ri].val/2^ARG_REDUC_1_SIZE */
uint8_t ri = (xbits >> (52 - ARG_REDUC_1_PREC))
             - (1u << ARG_REDUC_1_PREC);
uint64_t y = ARG_REDUC_1_GETVALUE(ri) * xbits;

/* Y = (1/S) * (1 + dZ),
 with dZ = dz/2^(52 + ARG_REDUC_1_SIZE + ARG_REDUC_2_SIZE)
 and 1/S = argReduc2[si].val/2^ARG_REDUC_2_SIZE */
uint8_t si = (y >> (52 + ARG_REDUC_1_SIZE - ARG_REDUC_2_PREC))
             - (1u << ARG_REDUC_2_PREC);
uint64_t dz = ARG_REDUC_2_GETVALUE(si) * y;
// the integer part of the fixed-point is removed by overflow
```

Why stop at two levels of reduction?

Answer is: diminushing return.

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

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

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"

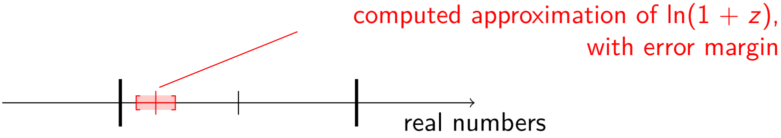
- Iterate on the same range reduction
- Stop as soon as Taylor at order 2 is good enough:
  $p(z) = z - z^2/2$ because it is very easy to compute
- Build ad-hoc rectangular multipliers
- No need to tabulate $1/(1 + x_i)$ when x_i is small enough.

Polynomial approximation (advertisement)

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:



Fixed-point means: explicit shifts

```
/* Polynomial approximation of log(1+Z)/Z ~= P(Z),
   and evaluate Z*P(Z) */
uint64_t p = UINT64_C(0xffffffffffffffc4)
               -(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);
```

Note that some of the shifts are inside the constants

Reconstructing the solution

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

Reconstructing the solution

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

Reconstructing the solution

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

Reconstructing the solution

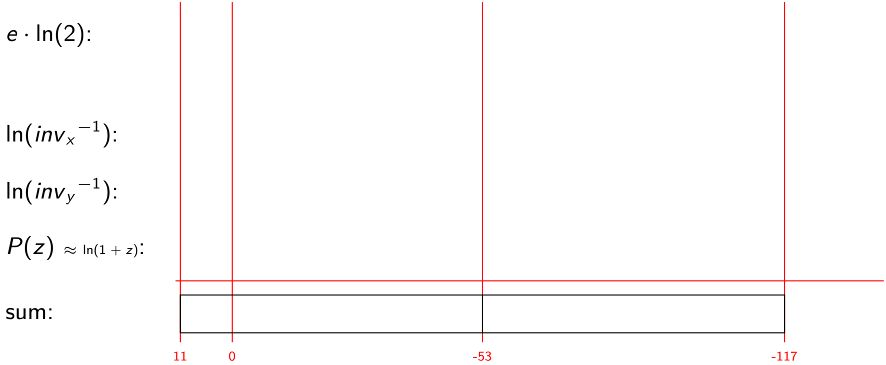
$$input = 2^e \cdot \frac{1}{inv_x} \cdot \frac{1}{inv_y} \cdot (1+z)$$

$$\ln(input) = e \cdot \ln(2) \quad + \quad \ln({inv_x}^{-1}) \quad + \quad \ln({inv_y}^{-1}) \quad + \quad \ln(1+z)$$

Reconstructing the solution

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

$$\ln(input) = e \cdot \ln(2) \quad + \quad \ln({inv_x}^{-1}) \quad + \quad \ln({inv_y}^{-1}) \quad + \quad \ln(1+z)$$

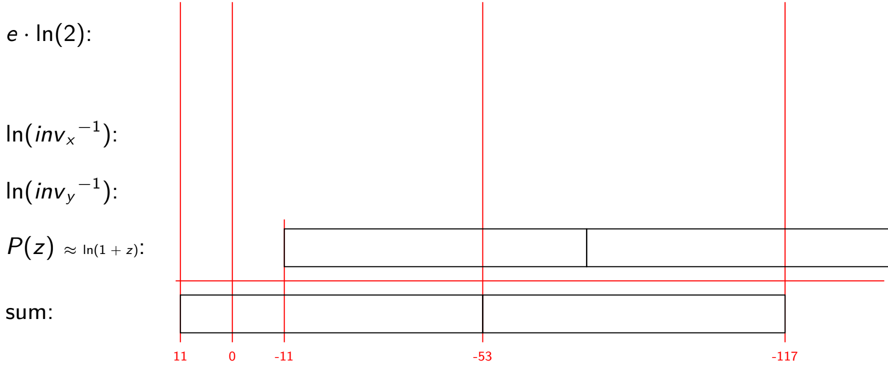


“If we can predict the exponents, exponent bits are wasted bits”

Reconstructing the solution

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

$$\ln(input) = e \cdot \ln(2) \quad + \quad \ln({inv_x}^{-1}) \quad + \quad \ln({inv_y}^{-1}) \quad + \quad \ln(1+z)$$

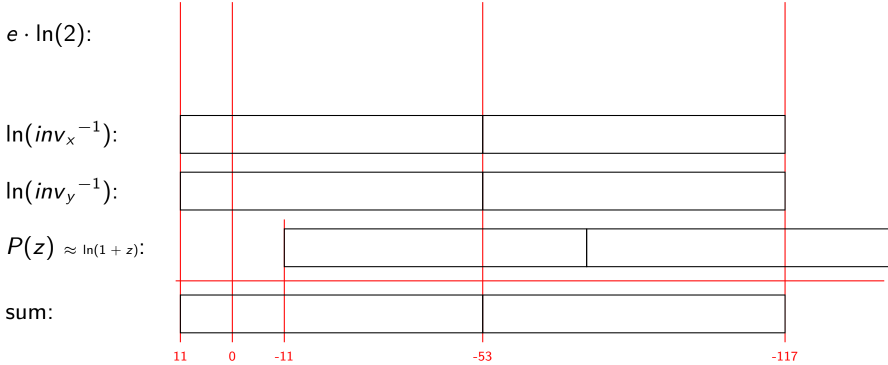


"If we can predict the exponents, exponent bits are wasted bits"

Reconstructing the solution

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

$$\ln(input) = e \cdot \ln(2) \quad + \quad \ln({inv_x}^{-1}) \quad + \quad \ln({inv_y}^{-1}) \quad + \quad \ln(1+z)$$

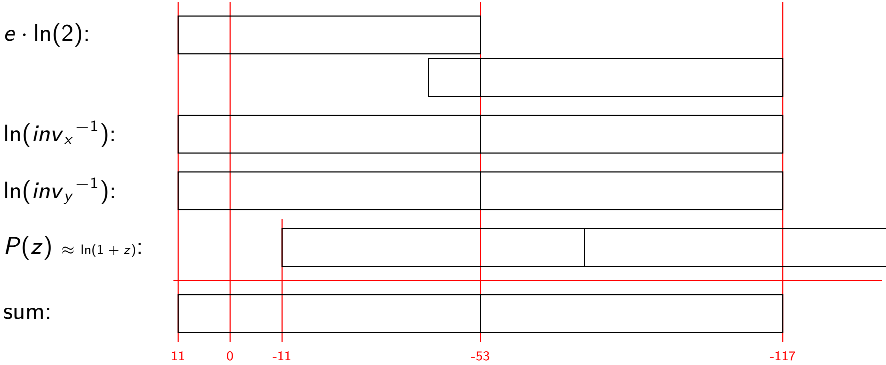


"If we can predict the exponents, exponent bits are wasted bits"

Reconstructing the solution

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

$$\ln(input) = e \cdot \ln(2) \quad + \quad \ln({inv_x}^{-1}) \quad + \quad \ln({inv_y}^{-1}) \quad + \quad \ln(1+z)$$

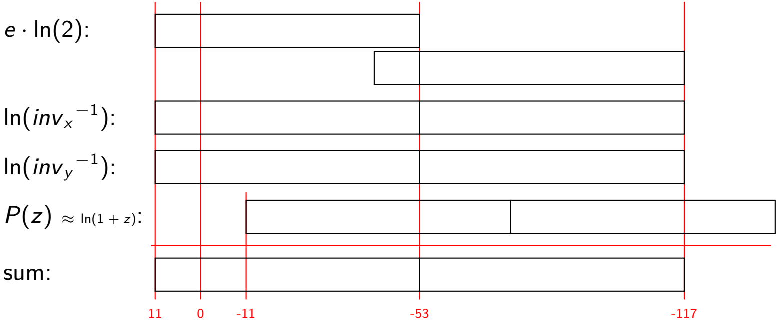


"If we can predict the exponents, exponent bits are wasted bits"

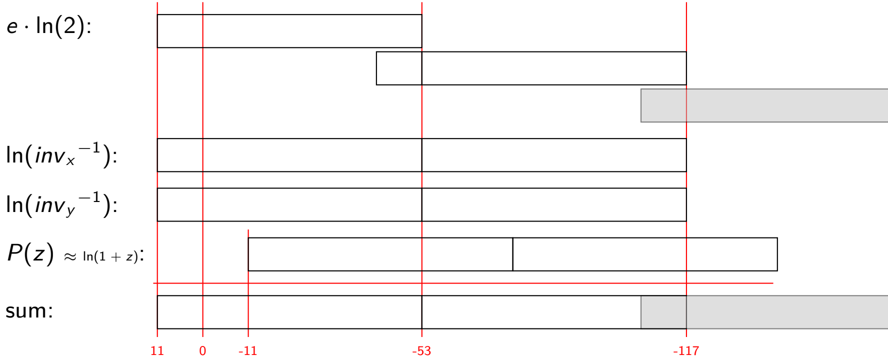
Now it really gets ugly

```
/* 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) // 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 >
// if sign != 0, this is longres = ~ longres: it approximate the a
// to avoid the approximation, do: longres = ((int64_t)sign + long
longres ^= UINT128(sign, sign);

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

Error evaluation

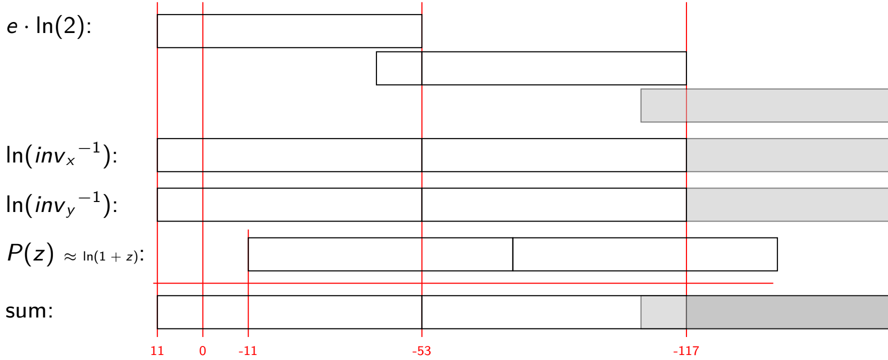


Error evaluation



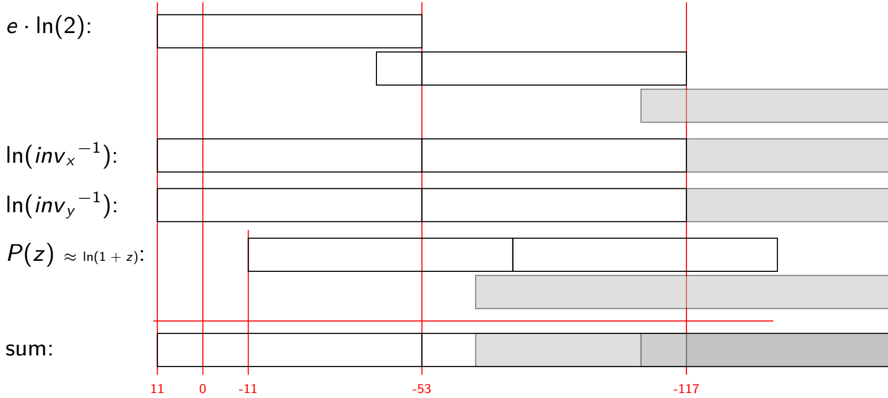
$$\epsilon < (|e|) \cdot 2^{-117}$$

Error evaluation



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

Error evaluation

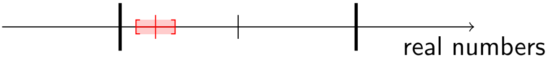


$$\epsilon < \left(|e| + 1 + 1 + P(z) \cdot 2^{-55}\right) \cdot 2^{-117}$$

Rounding test

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

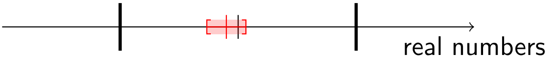
(two additions, a xor and a shift)



Rounding test

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

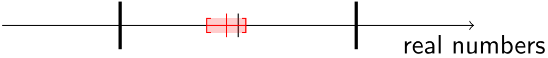
(two additions, a xor and a shift)



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...

Error evaluation and rounding test

```
/* Compute the maximal absolute error (aligned with longres)
 If result*(1 +- maxRelErr) are not rounded to the same number, we
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 */
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;
```

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

Outline

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

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

cycles	*MKL*	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

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

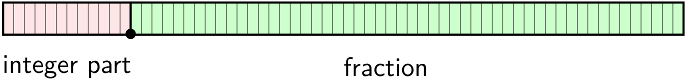
- dynamic programming algorithm:
  alignment as a path within a 2D array.
- borders of an array initialized with log-likelihoods
- then array filled using recurrence formulae
  that involve only max and + operations.

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



- faithful: 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.

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

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

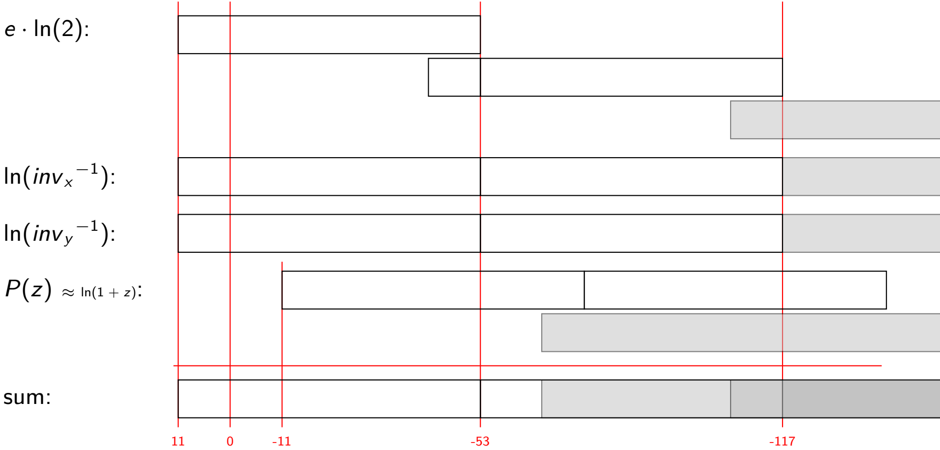
Going further with the fixed-point arithmetic

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

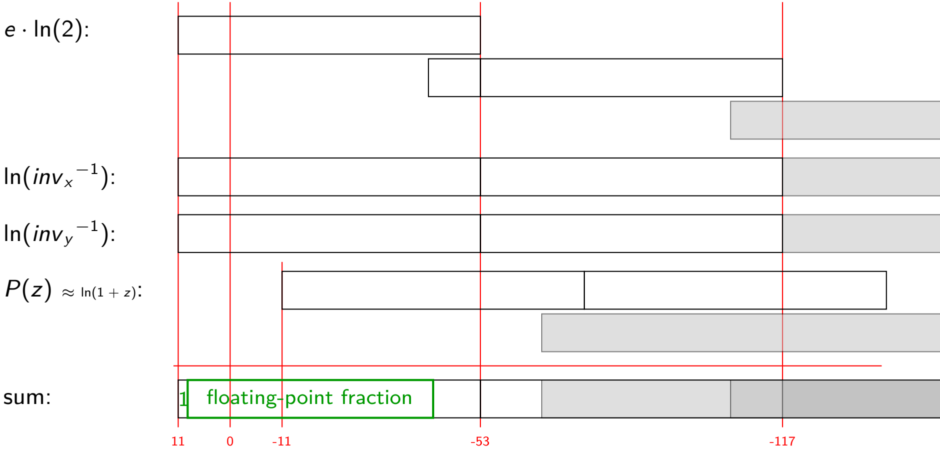
Thanks for your attention

Any question ?

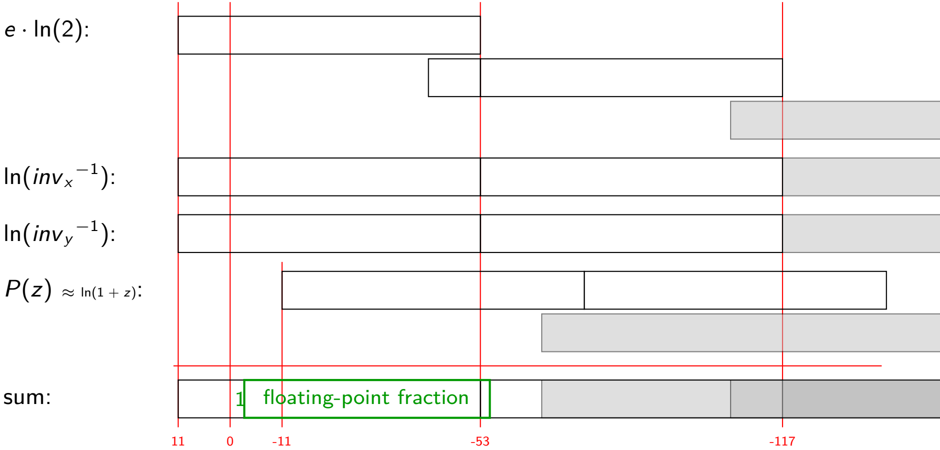
Reconstructing the solution



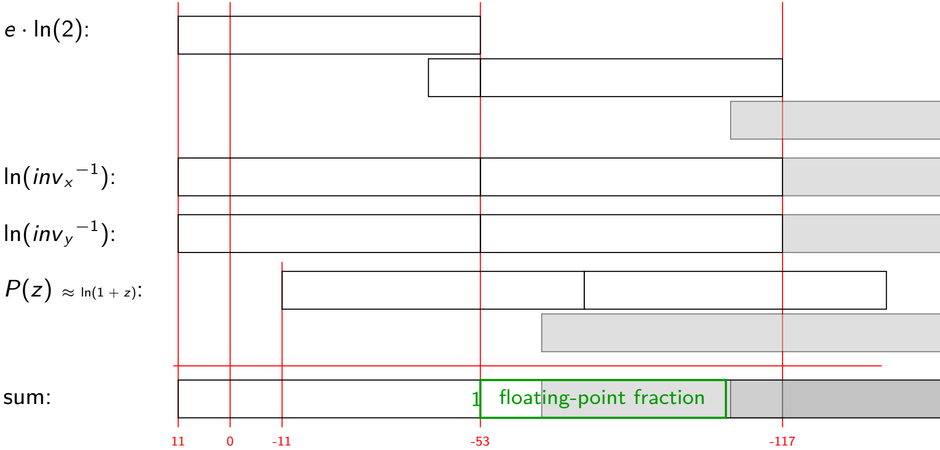
Reconstructing the solution



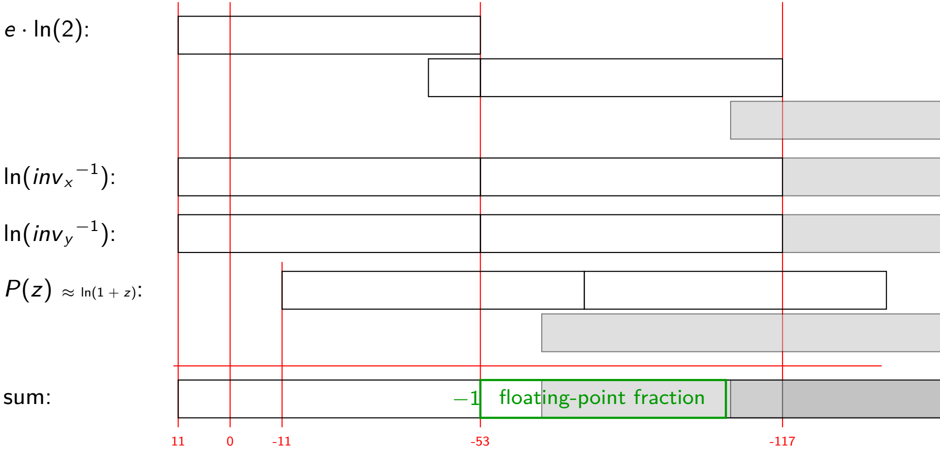
Reconstructing the solution



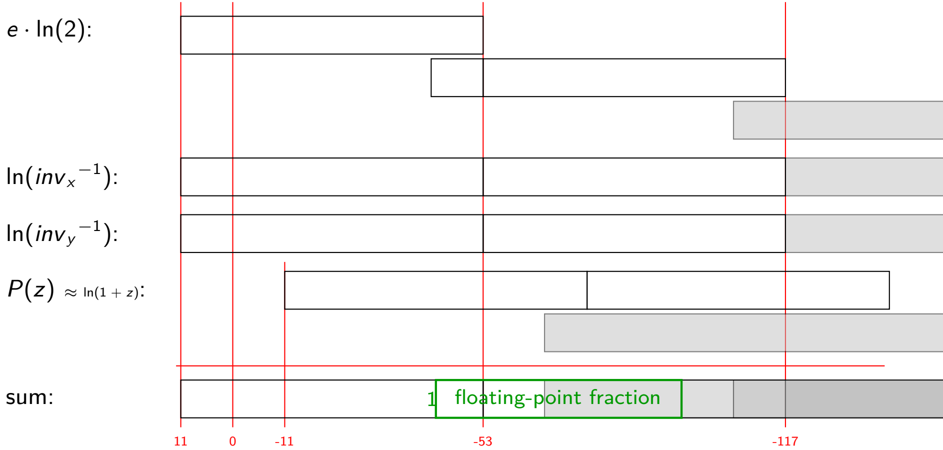
Reconstructing the solution



Reconstructing the solution



Reconstructing the solution



Example of code 2

```
/* X = 2^xe * (xbits/2^52) */
xe -= 1023;
xbits = (xbits & 0xFFFFFFFFFFFFFull) + (UINT64_C(1) << 52);

/* X = 2^xe * (1/R) * Y,
 with  Y = y/2^(52 + ARG_REDUC_1_SIZE)
 and 1/R = argReduc1[ri].val/2^ARG_REDUC_1_SIZE */
uint8_t ri = (xbits >> (52 - ARG_REDUC_1_PREC)) - (1u << ARG_REDUC_1_PREC);
uint64_t y = ARG_REDUC_1_GETVALUE(ri) * xbits;

/* Y = (1/S) * (1 + dZ),
 with dZ = dz/2^(52 + ARG_REDUC_1_SIZE + ARG_REDUC_2_SIZE)
 and 1/S = argReduc2[si].val/2^ARG_REDUC_2_SIZE */
uint8_t si = (y >> (52 + ARG_REDUC_1_SIZE - ARG_REDUC_2_PREC)) - (1u << ARG_REDUC_2_PREC);
uint64_t dz = ARG_REDUC_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) // dont 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(0xffffffffffffffc4)
             -(highmul(dz,
                 UINT64_C(0x7fffffffff091895)
                 -(highmul(dz,
                     UINT64_C(0x55555509230fb34c)
                     -(highmul(dz,UINT64_C(0x3ff8f2ad563f0e19))>>IMPLICIT_ZEROS)
                  )>>IMPLICIT_ZEROS)
              )>>IMPLICIT_ZEROS);
uint128_t zpzpart = fullmul(dz, p);
```

Example of code 3

```
/* 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, and ~0 otherwise
// if sign != 0, this is longres = ~ longres: it approximate the absolute value (-a = ~a + 1)
// 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);

/* Compute the maximal absolute error (aligned with longres)
 If result*(1 +- maxRelErr) are not rounded to the same number, we need more precision */
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 */
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;
```