Approximating at Scale

How string to float in LLVM’s libc is faster

Who am I?

- I’ve been working at Google for ~2 years.
- I’ve been working on LLVM’s libc for ~1 year.
- I’m currently in charge of the string functions.
    - This included writing the strtof (and related functions) implementation

Other fun facts:

- I graduated from RPI in 2020
- I play video games in my free time
    - Mostly Dota 2 and Beat Saber recently

Michael Jones

What does string to float conversion do?

Input:

“3.1415”

“11235813213455.89”

“6.022e23”

“20040229”

“36e529”

Output:

IEEE 754 Double-Precision Floating-Point Value

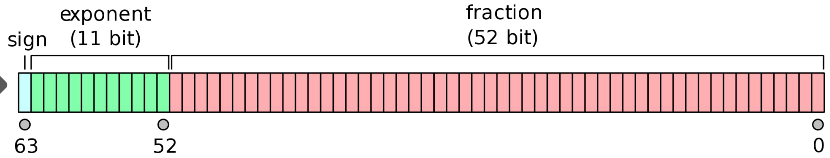


Image credit:

IEEE 754 double precision by Codekaizen - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=3595583

The Simple Version

Input numbers:

$\text{Digits} \times 10^{\text{Starting Exponent}}$

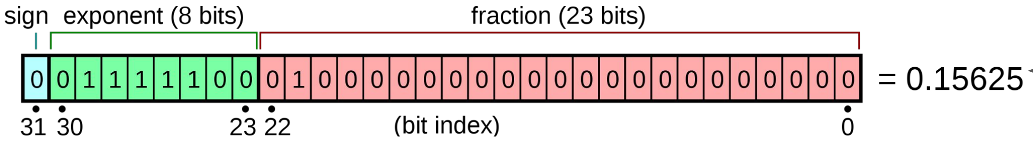
Output numbers:

$\text{Mantissa} \times 2^{\text{Final Exponent}}$

$1 <= \text{Mantissa} < 2$

The complex version

IEEE 754 defines how floating point numbers are represented:



Sign bit: 0 for positive, 1 for negative (in blue)

Exponent: a biased integer that shifts the number up and down (in green)

Mantissa (fraction): The number after the decimal point (in red)

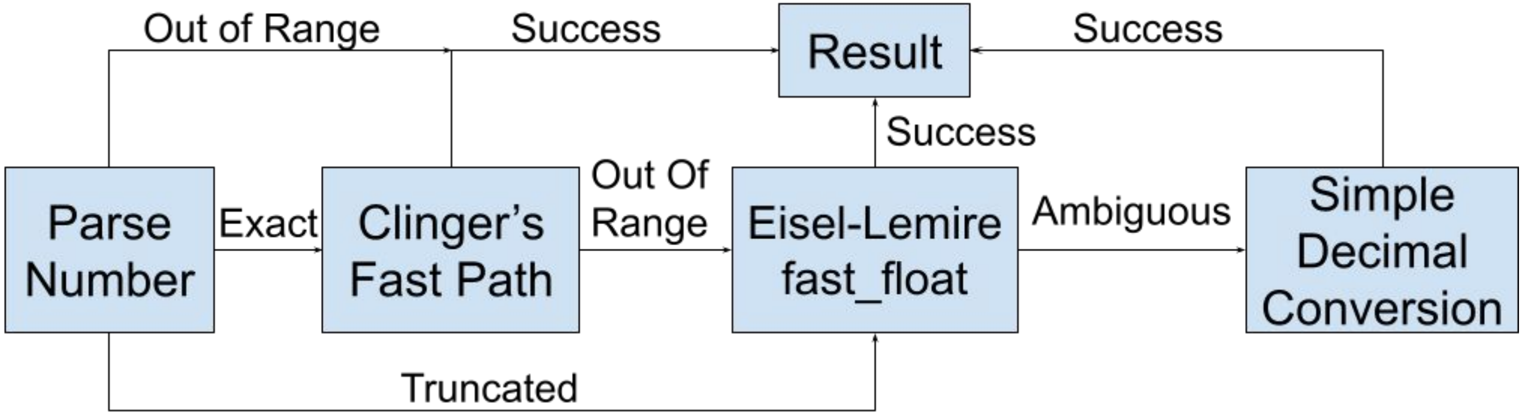
Value = $(-1)^{\text{sign}} \times 2^{(\text{exponent - bias})} \times 1.\text{mantissa}$

Example: $(-1)^{0} \times 2^{(124 - 127)} \times \text{0b1.01} = 2^{(-3)} \times 1.25 = 0.15625$

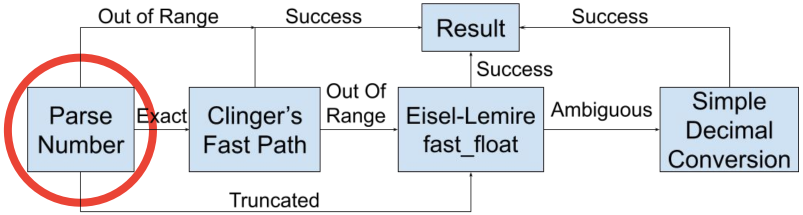
Image Credit: IEEE 754 single precision By Vectorization: Stannered - Own work based on: Float example.PNG, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=3357169

How do we make it faster?

Instead of improving the high accuracy algorithm, we use multiple passes to evaluate the most common numbers faster.



Parsing just the important parts


Parse Number
Clinger's Fast Path
Eisel-Lemire fast_float
Simple Decimal Conversion
Result
Out of Range
Success
Success
Success
Exact
Out Of Range
Ambiguous
Truncated

Exact

Integer = 31415

3.1415

Base 10 Exp = -4

Truncated

Integer = 1123581321

11235813213455.89

Base 10 Exp = 4

Out Of Range

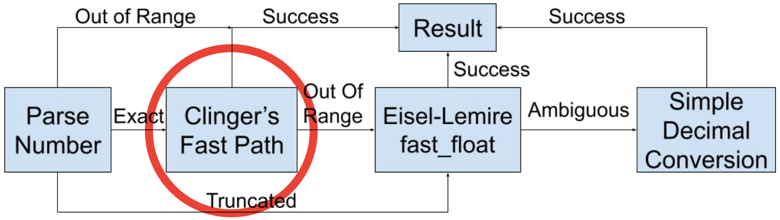
Integer = 36

36e529

Base 10 Exp = 529

Result = inf

First pass: Clinger’s Fast Path



Success

Integer = 31415

Base 10 Exp = -4

Result = 31415 / 10^4

= 3.1415

In Range for Double

Integer = 6022

Base 10 Exp = 20

Result = 6022 * 10^{20}

= 6.022e23

Out of Range For Float

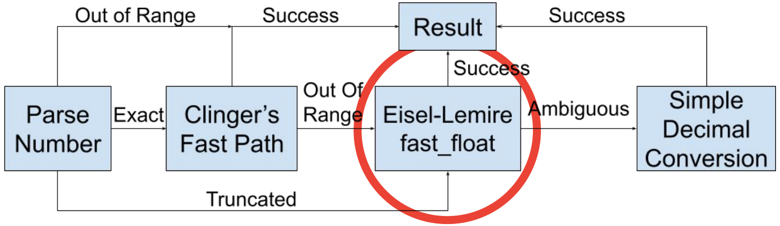
Integer = 75

Base 10 Exp = 17

float(10^{17}) =

99999998430674944

Second Pass:
Eisel-Lemire fast_float



Integer = 75

Base 10 Exp = 17   Base 2 Exp = 17 × $\log_2(10)$

Result = Leftshift(75) × Power Of Ten

Table[17]

= 0x96000000 × 0xB1A2BC2EC5000000

= 0x68155a43676e000000000000

= 0x1.a05569 × 2^{62}

Integer = 20040229

Base 10 Exp = 0

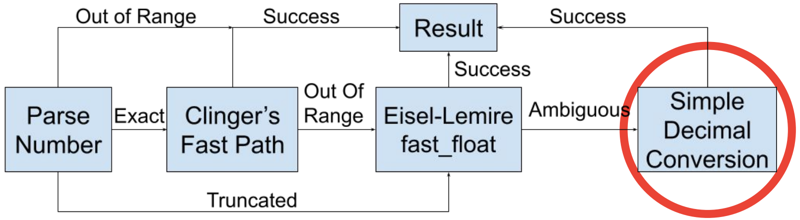
Result = Leftshift(20040229) × Table[0]

= 0x98e51280 × 0x8000000000000000

= 0x4c7289400000000000000000

Exactly halfway between 20040230 and 20040228

Third Pass:
Simple Decimal Conversion



Out of Range

Success

Result

Success

Success

Parse Number

Exact

Clinger's Fast Path

Out Of Range

Eisel-Lemire fast_float

Ambiguous

Simple Decimal Conversion

Truncated

$\text{Digits} \times 10^{\text{Starting Exponent}} = \text{Mantissa} \times 2^{\text{Final Exponent}}$

And $1 <= \text{Mantissa} < 2$

Two variables, two equations (and inequalities), we can just do the algebra.

Pro: Can always get the correct answer.

Con: Have to store $\text{Digits} \times 10^{\text{Starting Exponent}}$ in a High Precision Decimal.

Now to test

I'm using [Nigel Tao's Parse Number FXX Test Data]() which has 5,299,993 test cases and a Xeon based workstation for my testing. I'm also using Clang 14.0.6-2 and Glibc 2.35 for comparison.

Here is the cmake command I use to set up a proper speed comparison:

```
cmake ../llvm -G Ninja -DLLVM_ENABLE_PROJECTS="llvm;libc" -DCMAKE_BUILD_TYPE=Release
-DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ -DLLVM_LIBC_FULL_BUILD=ON
-DLLVM_LIBC_INCLUDE_SCUDO=OFF -DLIBC_COMPILE_OPTIONS_DEFAULT="-O3"
```

The ninja targets for testing this are **`libc_str_to_float_comparison_test`** for my implementation and **`libc_system_str_to_float_comparison_test`** for your system libc. To run them on the test data, build the targets and run them like so:

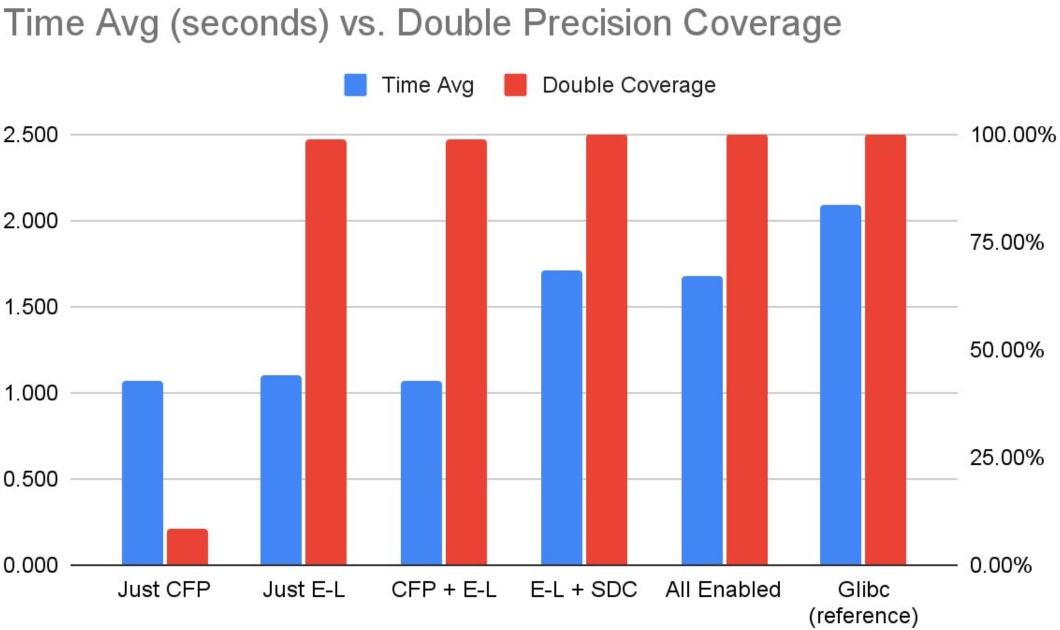
```
time ~/llvm-project/build/bin/libc_str_to_float_comparison_test ~/parse-number-fxx-test-data/data/*
```

How fast is it?

	Seconds
LLVM Libc	1.676
Glibc 2.35	2.087
% improvement	19.70%

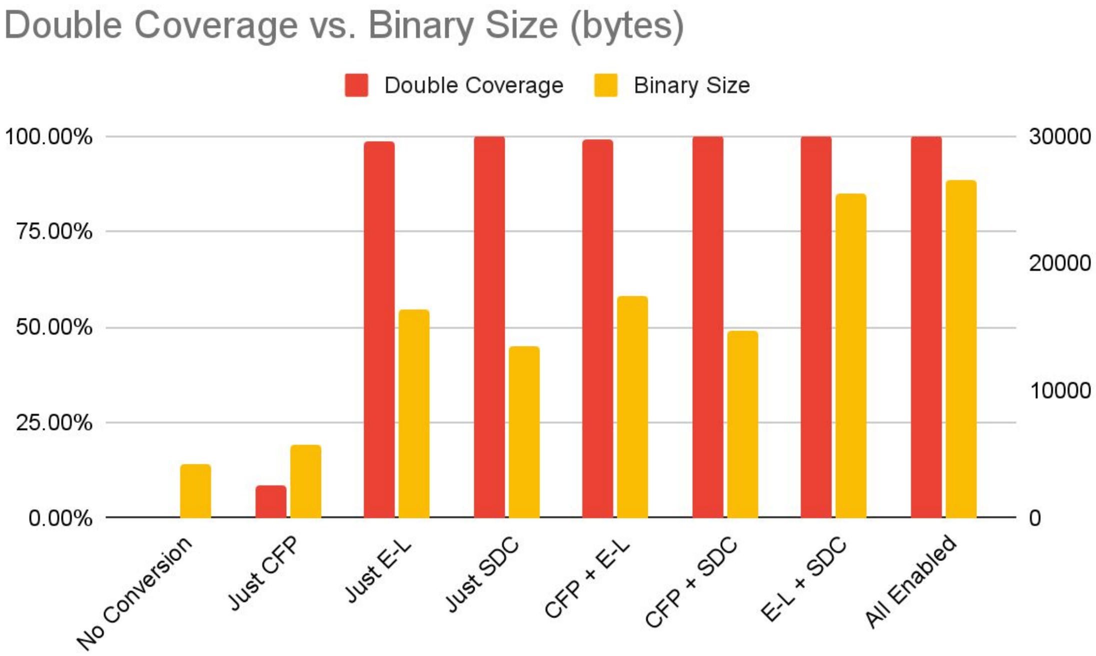
The Juicy Details

	Time Avg	Double Coverage
Just CFP	1.064	8.41%
Just E-L	1.097	98.63%
Just SDC	201.313	100.00%
CFP + E-L	1.071	98.99%
CFP + SDC	200.356	100.00%
E-L + SDC	1.708	100.00%
All Enabled	1.676	100.00%
Glibc 2.35	2.087	100.00%



At what cost?

	Double Coverage	Binary Size
No Conversion	0.04%	4248
Just CFP	8.41%	5720
Just E-L	98.63%	16408
Just SDC	100.00%	13472
CFP + E-L	98.99%	17464
CFP + SDC	100.00%	14712
E-L + SDC	100.00%	25552
All Enabled	100.00%	26584



Citations

- Clinger WD. How to Read Floating Point Numbers Accurately. SIGPLAN Not 1990 Jun;25(6):92–101. https://doi.org/10.1145/93548.93557.
    - William D. Clinger’s paper that includes Clinger’s Fast Path.
- Number Parsing at a Gigabyte per Second, Software: Practice and Experience 51 (8), 2021 (https://arxiv.org/abs/2101.11408)
    - Daniel Lemire’s paper describing the fast_float algorithm.
- https://nigeltao.github.io/blog/2020/eisel-lemire.html
    - Nigel Tao’s page explaining more of why the fast_float algorithm works.
- https://nigeltao.github.io/blog/2020/parse-number-f64-simple.html
    - Nigel Tao’s description of the Simple Decimal Conversion algorithm.
- https://github.com/nigeltao/parse-number-fxx-test-data
    - Nigel Tao’s test data set.