SHORT COMMUNICATION

Fast Number Parsing Without Fallback

Noble Mushtak[1*] | **Daniel Lemire**[2†]

[1]Northeastern University, Boston, MA, United States

[2]DOT-Lab Research Center, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada

Correspondence
Daniel Lemire, DOT-Lab Research Center, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada
Email: lemire@gmail.com

Funding information
Natural Sciences and Engineering Research Council of Canada, Grant Number: RGPIN-2017-03910

In recent work, Lemire (2021) presented a fast algorithm to convert number strings into binary floating-point numbers. The algorithm has been adopted by several important systems: e.g., it is part of the runtime libraries of GCC 12, Rust 1.55, and Go 1.16. The algorithm parses any number string with a significand containing no more than 19 digits into an IEEE floating-point number. However, there is a check leading to a fallback function to ensure correctness. This fallback function is never called in practice. We prove that the fallback is unnecessary. Thus we can slightly simplify the algorithm and its implementation.

KEYWORDS
Parsing, IEEE-754, Floating-Point Numbers

1 | INTRODUCTION

Current computers typically support 32-bit and 64-bit IEEE-754 binary floating-point numbers in hardware [1]. Real numbers are approximated by binary floating-point numbers: a fixed-width integer m (the *significand*) multiplied by 2 raised to an integer exponent p: $m \times 2^p$. Numbers are also frequently exchanged as strings in decimal form (e.g., `3.1416`, `1.0e10`, `4E3`). Given decimal numbers in a string, we must find efficiently the nearest available binary floating-point numbers when loading data from text files (e.g., CSV, XML or JSON documents).

A 64-bit binary floating-point number relies on a 53-bit significand m. A 32-bit binary floating-point number has 24-bit significand m. Given a string representing a non-zero number (e.g., `-3.14E+12`), we represent it as a sign (e.g., -), an integer $w \in [1, 2^{64})$ (e.g., $w = 314$) and a decimal power (e.g., 10): -314×10^{10}. The smallest positive value that can be represented using a 64-bit floating-point number is 2^{-1074}. We have that $w \times 10^{-343} < 2^{-1074}$ for all $w < 2^{64}$. Thus if the decimal exponent is smaller than -342, then the number must be considered to be zero. If the decimal exponent is greater than 308, the result must be infinite (beyond the range).

In the simplest terms, to represent a decimal number into a binary floating-point number, we need to multiply the decimal significand by a power of five: $m \times 10^q = (m \times 5^q) \times 2^q$. Or, conversely, divide it by a power of five. For large

	Intel Ice Lake, GCC 11	Apple M2, LLVM 14
base instructions per number	271	299
improved instructions per number	257	295
CPU cycles per number	57.2	44.6
improved CPU cycles per number	55.5	43.0

TABLE 1 Performance comparison while parsing the numbers of the canada dataset [2] using CPU performance counters. The reference is the `fast_float` library (version 3.2.0). We estimate a 2% error margin on the cycle/number metric while the instruction count is nearly error-free. We get the improved numbers by removing the unnecessary check.

powers, an exact computation is impractical thus we use truncated tables.

Lemire's approach to compute the binary significand is given in Algorithm 1 [2]: the complete algorithm needs to compute the binary exponent, handle subnormal numbers, and rounding. The algorithm effectively multiplies the 64-bit decimal significand with a 128-bit truncated power of five, or the reciprocal of a power of five, conceptually producing a 192-bit product but truncating it to its most significant 128 bits. When the last 102 bits of the 128-bit product (for 32-bit floating-point numbers) or the last 73 bits of the 128-bit product (for 64-bit floating-point numbers) is made entirely of 1 bits, a more accurate computation (e.g., relying on the full power of five) may produce a different result. Thus the algorithm may sometimes fail and require a fallback (line 4). However, Lemire did not produce an example where a fallback is needed. We want to show that it never happens for 32-bit and 64-bit floating-point numbers and that the algorithm always succeeds. Hence, the check and fallback are unnecessary. The check represents a small computational cost. Table 1 presents the number of instructions and cycles per number in one dataset for two systems.[1] By removing the check, we reduce the number of instructions and CPU cycles per number parsed by 5% on one system (Intel) and by slightly over 1% on another (Apple).

Algorithm 1 Algorithm to compute the binary significand from a positive decimal floating-point number $w \times 10^q$ for the IEEE-754 standard. We compute more bits to allow for exact rounding and to account for a possible leading zero bit. We use the convention that a mod b is the remainder of the integer division of a by b.

Require: an integer $w \in [1, 2^{64})$ and an integer exponent $q \in (-342, 308)$

Require: a table T containing 128-bit reciprocals and powers of five for powers from -342 to 308 ▷ [2, Appendix B]

1: $l \leftarrow$ the number of leading zeros of w as a 64-bit (unsigned) word

2: $v \leftarrow 2^l \times w$ ▷ We normalize the significand.

3: Compute the 128-bit truncated product $z \leftarrow (T[q] \times v) \div 2^{64}$.

4: **if** (z mod $2^{73} = 2^{73} - 1$ for 64-bit numbers or z mod $2^{102} = 2^{102} - 1$ for 32-bit numbers) and $q \notin [-27, 55]$ **then**

5: **Fallback needed** ▷ [2, Remark 1]

6: **end if**

7: **Return** $m \leftarrow$ the most significant 55 bits (64-bit) or 26 bits (32-bit) of the product z

[1] https://github.com/lemire/simple_fastfloat_benchmark

2 | RELATED WORK

Clinger [3, 4] was maybe earliest in describing accurate and efficient decimal-to-binary conversion techniques. He proposed a fast path using the fact that small powers of 10 can be represented exactly as floats. His fast path is still useful today. Gay [5] implemented a fast general decimal-to-binary implementation that is still popular. Gay's strategy is to first find quickly a close approximation, and then to refine it with exact big-integer arithmetic.

The reverse problem, binary-to-decimal conversion, has received much attention [6]. Adams [7, 8] bound the maximum and minimum of $ax \bmod b$ over an interval starting at zero, to show that powers of five truncated to 128 bits are sufficient to convert 64-bit binary floating-point numbers into equivalent decimal numbers.

3 | CONTINUED FRACTIONS

Given a sequence of integers $a_0, a_1, \ldots$, the expression $a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots}}}$ is a *continued fraction*. We can write a rational number n/d where n and $d > 0$ are integers as a continued fraction by computing $a_0, a_1, \ldots$. A continued fraction represents a sequence of converging values $c_0 = a_0, c_1 = a_0 + \frac{1}{a_1}, c_2 = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2}}, c_3 = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3}}}, \ldots$ We call each such value $(c_0, c_1, c_2, \ldots)$ a *convergent*. Each convergent is a rational number: we can find integers p_n and q_n such that $c_n = p_n/q_n$. We are interested in computing the convergents quickly. We can check that $p_0 = a_0, q_0 = 1$, $p_1 = a_1 p_0 + 1, q_1 = a_1 q_0$, and $p_2 = a_2 p_1 + p_0, q_2 = a_2 q_1 + q_0$. We have the general formulas $p_n = a_n p_{n-1} + p_{n-2}, q_n = a_n q_{n-1} + q_{n-2}$. These recursion formulas were known to Euler. Rational numbers that are close to a value x are also convergents according to the following theorem due to Legendre [9, 10].

Theorem 1 *For $q > 0$ and p, q integers, if the fraction p/q is such that $|p/q - x| < \frac{1}{2q^2}$, then p/q is a convergent of x.*

4 | RULING OUT FALLBACK CASES

Let w and q be the decimal significand and the decimal exponent, respectively, of the decimal number to be converted into a binary floating-point number. We require that the decimal significand is a positive 64-bit number. We multiply w by a power of two to produce $v \in [2^{63}, 2^{64})$. Let $T[q]$ be the 128-bit number representing a truncated power of five or the reciprocal of a power of five. The fallback condition may be needed when the least significant 73 bits of the most significant 128 bits of the product of v and $T[q]$ are all 1 bits. When parsing 32-bit floating-point numbers, we have a more generous margin (102 bits instead of 73 bits), so it is sufficient to review the 64-bit case.

Assume that the least significant 73 bits of the most significant 128 bits of the product of v and $T[q]$ are all 1 bits. It is equivalent to requiring that the number made of the least significant 137 bits of the product, $r = (T[q] \times v) \bmod 2^{137}$, is larger than or equal to $2^{137} - 2^{64}$: $r \geq 2^{137} - 2^{64}$. By the next lemma, we have that v is the denominator of a convergent of $T[q]/2^{137}$.

Lemma 2 *Given positive integers $T[q] < 2^{128}$ and $v < 2^{64}$, we have that $(T[q] \times v) \bmod 2^{137} \geq 2^{137} - 2^{64}$ implies that n/v, for $n = 1 + \lfloor T[q] \times v/2^{137} \rfloor$, is a convergent of $T[q]/2^{137}$.*

Proof By definition, we have that $a = b\lfloor a/b \rfloor + (a \bmod b)$ given two integers a, b when $b > 0$. Thus, given $r = (T[q] \times v) \bmod 2^{137}$, we have that $(T[q] \times v) = 2^{137}\lfloor T[q] \times v/2^{137} \rfloor + r$. Substracting r from both sides, we get $(T[q]\times v) - r = 2^{137}\lfloor T[q]\times v/2^{137} \rfloor$. Adding 2^{137} on both sides, we get $(T[q]\times v) + 2^{137} - r = 2^{137} + 2^{137}\lfloor T[q]\times v/2^{137} \rfloor =$

$2^{137}n$. Let $x = 2^{137} - r$, then we have $(T[q] \times v) + x = 2^{137}n$. And because v is non-zero we can divide by $v \times 2^{137}$ throughout to get $\frac{T[q]}{2^{137}} + \frac{x}{v \times 2^{137}} = n/v$.

Because $(T[q] \times v) \bmod 2^{137} \geq 2^{137} - 2^{64}$, we have that $r \geq 2^{137} - 2^{64}$, and so $2^{137} - r \leq 2^{64}$ or $x \leq 2^{64}$.

Finally, we have

$$\begin{aligned}
&\left|\frac{T[q]}{2^{137}} - \frac{n}{v}\right| = \frac{x}{v \times 2^{137}} \\
\Rightarrow \quad &\leq \frac{1}{v \times 2^{73}} && \text{because } x \leq 2^{64} \\
\Rightarrow \quad &< \frac{1}{2v^2} && \text{because } v < 2^{64}.
\end{aligned}$$

Hence, we have that n/v is a convergent of the rational number $T[q]/2^{137}$ by Theorem 1. □

Thus, if the fallback condition is triggered by some decimal significand v and some decimal scale q, the significand v must be the denominator of a convergent of $T[q]/2^{137}$.

It remains to show that it suffices to check the convergent as simple fractions. If n/v is a convergent, then $(n/d)/(v/d)$ where $d = \gcd(n, d)$ is the simple counterpart. If $v < 2^{64}$, then $v/d < 2^{64}$. We want to show that if $(T[q] \times v) \bmod 2^{137} \geq 2^{137} - 2^{64}$ then $(T[q] \times \frac{v}{d}) \bmod 2^{137} \geq 2^{137} - 2^{64}$.

Consider the following technical lemma which relates $a \times \frac{v}{d} \bmod 2^{137}$ and $(a \times v) \bmod 2^{137}$.

Lemma 3 *Given positive integers a and v, we have that $\left(a \times \frac{v}{d}\right) \bmod 2^{137} = 2^{137} + \frac{(a \times v) \bmod 2^{137} - 2^{137}}{d}$ where $d = \gcd(v, 1 + \lfloor a \times v/2^{137} \rfloor)$.*

Proof We have that $a \times v = 2^{137}(1 + \lfloor a \times v/2^{137} \rfloor) + ((a \times v) \bmod 2^{137}) - 2^{137}$. Given $d = \gcd(v, 1 + \lfloor a \times v/2^{137} \rfloor)$, then $(1 + \lfloor a \times v/2^{137} \rfloor)/d$ and v/d are integers. We have that $a \times v/d = 2^{137}(1 + \lfloor a \times v/2^{137} \rfloor)/d + ((a \times v) \bmod 2^{137} - 2^{137})/d$. Thus it follows that $(a \times v/d) \bmod 2^{137} = 2^{137} + ((a \times v) \bmod 2^{137} - 2^{137})/d$. □

Suppose that $(T[q] \times v) \bmod 2^{137} \geq 2^{137} - 2^{64}$, then by Lemma 3, we have

$$\begin{aligned}
\left(T[q] \times \frac{v}{d}\right) \bmod 2^{137} &= 2^{137} + \frac{(T[q] \times v) \bmod 2^{137} - 2^{137}}{d} \\
&\geq 2^{137} + \frac{2^{137} - 2^{64} - 2^{137}}{d} \\
&\geq 2^{137} - 2^{64}/d.
\end{aligned}$$

Hence we have that if $(1 + \lfloor T[q] \times v/2^{137} \rfloor)/v$ is a convergent such that $(T[q] \times v) \bmod 2^{137} \geq 2^{137} - 2^{64}$, then the simplified convergent, $v' \leftarrow v/d$, also satisfies $(T[q] \times v') \bmod 2^{137} \geq 2^{137} - 2^{64}$ since $d \geq 1$. Thus it is sufficient to check the convergents in simplified form. We have proven the following proposition.

Proposition 4 *Given a positive integer $T[q]$, we have that there exists a positive integer $v < 2^{64}$ such that $(T[q] \times v) \bmod 2^{137} \geq 2^{137} - 2^{64}$ if and only if there is a continued-fraction convergent p'/q' of $T[q]/2^{137}$ such that p'/q' is a simple fraction ($\gcd(p', q') = 1$), $q' < 2^{64}$ and $(T[q] \times q') \bmod 2^{137} \geq 2^{137} - 2^{64}$.*

For any decimal exponent q, to check whether there exists a decimal significand $1 \leq v < 2^{64}$ which triggers the fallback condition, it suffices to examine all convergents of $T[q]/2^{137}$ with a denominator less than 2^{64} and

check whether $(T[q] \times v) \bmod 2^{137} \geq 2^{137} - 2^{64}$. We can convert $T[q]/2^{137}$ into a continued fraction and compute the coefficients $a_0, a_1, a_2, \ldots$. We can then use the recursive formulas for computing convergents to enumerate all convergents as simple fractions [9]. Given convergents as simple fraction $p_0/q_0, p_1/q_1, \ldots$, we check whether $(T[q] \times q_i) \bmod 2^{137} \geq 2^{137} - 2^{64}$ for $i = 0, 1, \ldots$ It is enough to check the convergents with a denominator smaller than 2^{64}. We implement this algorithm with a Python script: the script reports that no fallback is needed. Hence no fallback is required for 64-bit significands in the number-parsing algorithm described by Lemire [2].

References

[1] IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985 1985;p. 1–20.

[2] Lemire D. Number parsing at a gigabyte per second. Software: Practice and Experience 2021;51(8):1700–1727. https://doi.org/10.1002/spe.2984.

[3] Clinger WD. How to Read Floating Point Numbers Accurately. SIGPLAN Not 1990 Jun;25(6):92–101. https://doi.org/10.1145/93548.93557.

[4] Clinger WD. How to Read Floating Point Numbers Accurately. SIGPLAN Not 2004 Apr;39(4):360–371. https://doi.org/10.1145/989393.989430.

[5] Gay DM, Correctly rounded binary-decimal and decimal-binary conversions; 1990. AT&T Bell Laboratories Numerical Analysis Manuscript 90-10.

[6] Steele GL, White JL. How to Print Floating-Point Numbers Accurately. SIGPLAN Not 2004 Apr;39(4):372–389. https://doi.org/10.1145/989393.989431.

[7] Adams U. Ryū: Fast Float-to-String Conversion. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI 2018, New York, NY, USA: Association for Computing Machinery; 2018. p. 270–282. https://doi.org/10.1145/3192366.3192369.

[8] Adams U. Ryu Revisited: Printf Floating Point Conversion. Proc ACM Program Lang 2019 Oct;3(OOPSLA). https://doi.org/10.1145/3360595.

[9] Hardy GH, Wright EM, et al. An introduction to the theory of numbers. Oxford university press; 1979.

[10] Legendre AM. Essai sur la theorie des nombres. chez Courcier, imprimeur-libraire pour les mathematiques, quai des Augustins; 1808.