Article

xjb: Fast Float to String Algorithm

Junbo Xiang and Tiejun Wang *

Posted Date: 1 April 2026

doi: 10.20944/preprints202511.1698.v2

Keywords: floating-point; printing; algorithm; performance

 doi:10.20944/preprints202511.1698.v2

Article

xjb: Fast Float to String Algorithm

Junbo Xiang and Tiejun Wang *

Chengdu University of Information Technology
* Correspondence: tjw@cuit.edu.cn; Tel.:+86-138-8090-0111

Abstract

Efficiently and accurately converting floating-point numbers to decimal strings is a critical challenge in numerical computation and data exchange. While existing algorithms like Ryū, Dragonbox, and Schubfach satisfy the Steele-White (SW) principle for accuracy, they often suffer from performance bottlenecks due to branch prediction failures and high-precision multiplication overhead. This paper presents a novel floating-point to string conversion algorithm called "xjb", an optimized variant of the Schubfach algorithm designed to deliver superior performance for IEEE754 single-precision (binary32) and double-precision (binary64) floating-point numbers. By minimizing instruction dependencies, reducing multiplication operations, mitigating branch prediction penalties and by utilizing the simd instruction set, xjb achieves significant performance gains. The algorithm features concise core implementation. Extensive benchmarking across diverse platforms, including AMD R7-7840H and Apple M1, demonstrates that xjb outperforms state-of-the-art algorithms in most scenarios while maintaining full compliance with the SW principle.

Keywords: floating-point; printing; algorithm; performance

1. Introduction

Floating-point to decimal string conversion is a fundamental operation in computer systems with widespread applications across numerous domains. From scientific computing and financial systems to web services and database management, the ability to efficiently and accurately convert binary floating-point representations into human-readable decimal strings is essential. Despite its apparent simplicity, this conversion problem presents significant challenges in balancing the competing demands of correctness, performance, and output compactness.

1.1. Background and Motivation

In 1990, Steele and White [1,2] published the seminal paper *How to Print Floating-Point Numbers Accurately*, which established the foundational principles for optimal floating-point printing algorithms, now widely known as the Steele-White (SW) principle. The SW principle comprises four key requirements:

- **Information preservation**: The printed result must be parseable back to the original floating-point number without loss of precision.
- **Minimum length**: The output string should be as short as possible while maintaining information preservation.
- **Correct rounding**: When multiple representations satisfy the first two criteria, the algorithm must correctly round to the nearest value, with ties broken by selecting the even value.
- **Left-to-right generation**: The output digits should be generated sequentially from the most significant to the least significant digit.

These principles ensure that floating-point numbers have a unique, well-defined decimal representation that is both human-readable and machine-parseable. Algorithms satisfying the SW principle

 doi:10.20944/preprints202511.1698.v2

guarantee that the conversion process is reversible and produces the shortest possible output, which is crucial for data exchange, serialization, and user interface display.

Over the past three decades, significant research efforts have been devoted to developing efficient algorithms that satisfy the SW principle. Early approaches, such as the **Dragon4** algorithm, provided correct results but suffered from performance limitations due to the use of arbitrary-precision arithmetic. Subsequent innovations introduced various optimizations:

- **dtoa.c** [3] is an improved version of Dragon4, written by Robert G. Burger and R. Kent Dybvig in 1996. The source code link is https://www.netlib.org/fp/dtoa.c.
- **Grisu3** [4] pioneered the use of precomputed powers of ten to avoid expensive arbitrary-precision operations, though it occasionally falls back to slower methods for certain inputs.
- **Errol** [5] improved upon Grisu3 by reducing the fallback rate through more precise error analysis.
- **Ryū** [6,7] achieved significant performance gains through careful instruction scheduling and lookup table optimizations, establishing a new baseline for high-performance conversion.
- **Schubfach** [8] introduced a novel approach based on the concept of "Schubfach" (pigeonhole principle), providing a compact and elegant solution with competitive performance.
- **Grisu-Exact** [9] eliminated the need for fallbacks entirely while maintaining high performance.
- **Dragonbox** [10] reduces the number of multiplications by sacrificing more branches.
- **yy_double** [11] explored alternative computational strategies to minimize the multiplication cost, but there were still a few branches.
- **uscale** [12–14] was proposed by Russ Cox and is used to enhance the performance of the floating-point number printing algorithm in the Go programming language.
- **zmij** [15] is based on yy_double and has undergone extensive code optimization.

Despite these advances, existing algorithms still face several performance challenges that limit their effectiveness in high-throughput scenarios:

1. **Branch prediction penalties**: Many algorithms rely heavily on conditional branches to handle different cases, leading to frequent branch mispredictions on modern pipelined processors.
2. **High-precision multiplication overhead**: The conversion process requires high-precision arithmetic operations, particularly multiplications involving large precomputed constants, which can be expensive on standard hardware.
3. **Instruction dependency chains**: Sequential dependencies between operations limit instruction-level parallelism and prevent efficient utilization of modern superscalar processors.
4. **Limited SIMD utilization**: Most existing algorithms do not exploit vector instruction sets (SIMD) that are now ubiquitous in contemporary processors.

These limitations motivate the need for a new approach that addresses these performance bottlenecks while maintaining full compliance with the SW principle.

1.2. Contributions

This paper presents **xjb**, a novel floating-point to string conversion algorithm that achieves superior performance through systematic optimization of the underlying computational structure. The xjb algorithm is derived from the Schubfach algorithm and incorporates insights from yy_double and Dragonbox, but introduces several key innovations:

1. **Reduced instruction dependencies**: By carefully restructuring the computation, xjb minimizes data dependencies between operations, enabling better instruction-level parallelism and improved pipeline utilization.
2. **Minimized multiplication operations**: The algorithm reduces the number of expensive high-precision multiplications required during conversion, significantly decreasing computational overhead.
3. **Mitigated branch prediction penalties**: Through branchless programming techniques and careful case analysis, xjb minimizes conditional branches that could lead to prediction failures.

 doi:10.20944/preprints202511.1698.v2

4. **SIMD instruction utilization**: The algorithm is designed to leverage SIMD instructions where applicable.
5. **Concise core implementation**: Despite its sophisticated optimizations, xjb maintains a compact and readable core implementation, facilitating adoption and maintenance.

The xjb algorithm supports both IEEE 754 single-precision (binary32) and double-precision (binary64) floating-point formats, which are the most widely used floating-point representations in modern computing. For simplicity, this paper uses *float* to refer to IEEE 754 binary32 and *double* to refer to IEEE 754 binary64.

1.3. Evaluation Overview

We conducted extensive benchmarking of xjb across diverse hardware platforms, including AMD R7-7840H and Apple M1 processors. Our evaluation demonstrates that xjb outperforms state-of-the-art algorithms in most scenarios while maintaining full compliance with the SW principle. The algorithm exhibits excellent portability and scalability, making it suitable for deployment across a wide range of systems from embedded devices to high-performance servers.

The implementation code for the xjb algorithm and all benchmark tools used in this paper are available at https://github.com/xjb714/xjb.

The remainder of this paper is organized as follows. Section 2 reviews the IEEE 754 floating-point representation and establishes the mathematical foundation for the conversion problem. Section 3 presents the core principles and derivation of the xjb algorithm and describes the implementation details and optimizations. Section 4 presents experimental results comparing xjb against existing algorithms.

1.4. Explanation of Special Symbols in This Article

Provide special explanations for the special symbols used in the formulas of this article.

Table 1. Explanation of Special Symbols in This Article.

symbol	brief explanation	example
%	Integer modulus operation	2 = 8%3
//	Integer division operation	1 = 5//3
<< or >>	Left or right shift of binary values	8 = 1<<3
? :	Similar to the ternary operator in C syntax	a = 1?a:b

2. IEEE754 Floating-Point Number Representation

This section establishes the mathematical foundation for floating-point number representation and defines the notation used throughout this paper. We focus on the IEEE 754 standard, which is the most widely adopted floating-point arithmetic standard in modern computing systems.

2.1. Scope and Assumptions

For clarity of presentation, we make the following simplifying assumptions:

- We consider only positive floating-point numbers, as negative numbers differ only by a leading minus sign.
- We exclude special values (zero, NaN, and infinity) from our analysis, as these are handled separately in practice.

These assumptions are standard in the literature and do not affect the generality of our algorithm, as the excluded cases can be handled with straightforward special-case logic.

2.2. Binary Representation

The IEEE 754 standard [16,17] defines two primary floating-point formats relevant to this work:

 doi:10.20944/preprints202511.1698.v2

Double-precision (binary64): A 64-bit format consisting of:

- 1 sign bit (s): indicates positive ($s = 0$) or negative ($s = 1$).
- 11 exponent bits (e): biased exponent in the range $[0, 2047]$.
- 52 fraction bits (f): significand fraction in the range $[0, 2^{52} - 1]$.

Single-precision (binary32): A 32-bit format consisting of:

- 1 sign bit (s): indicates positive ($s = 0$) or negative ($s = 1$).
- 8 exponent bits (e): biased exponent in the range $[0, 255]$.
- 23 fraction bits (f): significand fraction in the range $[0, 2^{23} - 1]$.

2.3. Classification of Floating-Point Numbers

We classify floating-point numbers into three categories based on their exponent and fraction fields:

1. **Subnormal numbers** ($e = 0$ and $f \neq 0$): These represent very small values close to zero, where the implicit leading bit of the significand is 0 instead of 1.
2. **Normal numbers** ($e \neq 0$ and $f \neq 0$): The standard case where the implicit leading bit of the significand is 1.
3. **Irregular numbers** ($e \neq 0$ and $f = 0$): Numbers with zero fraction field, representing powers of two.

We use the term **regular** to refer to both subnormal and normal numbers (i.e., all cases where $f \neq 0$).

2.4. Value Representation

The real value v of a positive floating-point number can be expressed in the unified form $v = c \cdot 2^q$, where c is the integer significand and q is the exponent. The general formula covering all cases is:

$$\begin{aligned} \text{double}: v &= \Big(f + (e \neq 0 \,?\, 2^{52} : 0)\Big) \cdot 2^{\max(e,1)-1075} = c \cdot 2^q \\ \text{float}: v &= \Big(f + (e \neq 0 \,?\, 2^{23} : 0)\Big) \cdot 2^{\max(e,1)-150} = c \cdot 2^q \end{aligned} \tag{1}$$

For each category, the values decompose as follows:

Subnormal numbers ($e = 0$):

$$\text{subnormal}: \begin{cases} \text{double}: & v = f \cdot 2^{-1074} \\ \text{float}: & v = f \cdot 2^{-149} \end{cases} \tag{2}$$

Irregular numbers ($f = 0, e \neq 0$):

$$\text{irregular}: \begin{cases} \text{double}: & v = 2^{52} \cdot 2^{e-1075} \\ \text{float}: & v = 2^{23} \cdot 2^{e-150} \end{cases} \tag{3}$$

Normal numbers ($e \neq 0, f \neq 0$):

$$\text{normal}: \begin{cases} \text{double}: & v = (f + 2^{52}) \cdot 2^{e-1075} \\ \text{float}: & v = (f + 2^{23}) \cdot 2^{e-150} \end{cases} \tag{4}$$

2.5. Rounding Intervals

A critical concept for accurate floating-point printing is the **rounding interval** R_v, which defines the range of real numbers that round to the given floating-point value v when parsed. The rounding interval is bounded by:

 doi:10.20944/preprints202511.1698.v2

$$
\begin{aligned}
v_l &= \begin{cases} \left(c - \frac{1}{2}\right) \cdot 2^q & \text{if } f \neq 0 \text{ or } e \leq 1 \\ \left(c - \frac{1}{4}\right) \cdot 2^q & \text{if } f = 0 \end{cases} \\
v_r &= \left(c + \frac{1}{2}\right) \cdot 2^q \\
R_v &= \begin{cases} [v_l, v_r] & \text{if } f \bmod 2 = 0 \text{ (even significand)} \\ (v_l, v_r) & \text{if } f \bmod 2 = 1 \text{ (odd significand)} \end{cases}
\end{aligned} \tag{5}
$$

The rounding radius for regular floating-point numbers is 2^{q-1}. The distinction between closed and open intervals at the boundaries depends on the parity of the significand, ensuring correct rounding according to the round-to-even rule specified in the IEEE 754 standard. Any decimal number within R_v will parse back to the original floating-point value v, which is essential for ensuring the information preservation property of the SW principle.

Table 2 summarizes the valid ranges for c and q across different categories.

Table 2. Valid ranges for significand c and exponent q

Category	Float (binary32)	Double (binary64)
Subnormal	$1 \leq c \leq 2^{23} - 1, q = -149$	$1 \leq c \leq 2^{52} - 1, q = -1074$
Normal	$2^{23} + 1 \leq c \leq 2^{24} - 1$	$2^{52} + 1 \leq c \leq 2^{53} - 1$
	$-148 \leq q \leq 104$	$-1073 \leq q \leq 971$
Irregular	$c = 2^{23}, -149 \leq q \leq 104$	$c = 2^{52}, -1074 \leq q \leq 971$

3. Principle of Algorithm

This section presents the algorithmic principles and mathematical foundation of the xjb floating-point to string conversion algorithm. We first introduce the overall architecture and design goals, followed by the mathematical formulation of the conversion problem.

3.1. Design Overview

This paper focuses on converting float (single-precision) and double (double-precision) floating-point numbers to decimal strings. The conversion process consists of two stages:

1. **Float-to-Decimal Conversion**: Converting binary floating-point values to decimal significand-exponent pairs (d, k).
2. **Decimal-to-String Conversion**: Formatting (d, k) into human-readable strings.

We focus primarily on the first stage, as it is the computationally intensive component that benefits most from our optimizations.

The xjb algorithm is designed with four key performance objectives:

- Minimize branch mispredictions through unlikely branches and branchless techniques
- Reduce expensive high-precision multiplication operations
- Optimize instruction dependencies to enable better parallelism
- Maintain core implementation simplicity and readability

3.2. Mathematical Foundation

Before presenting the algorithm details, we establish the mathematical framework for the float-to-decimal conversion problem.

Recall from Section 2 that any floating-point value v can be expressed in the form $v = c \cdot 2^q$, where c is the integer significand and q is the exponent. Our goal is to find the optimal decimal representation $opt = d \cdot 10^k$ that satisfies the SW principle.

 doi:10.20944/preprints202511.1698.v2

As established in Section 2, regular floating-point numbers (which include all normal and subnormal numbers with non-zero fraction fields) account for the vast majority of possible floating-point values. For the purposes of algorithm derivation, we focus primarily on regular numbers, as special cases can be handled with minimal additional logic.

The valid ranges for the significand c and exponent q for regular floating-point numbers are:

$$\begin{aligned} \text{float}: &\begin{cases} 1 \le c \le 2^{24}-1, c \ne 2^{23}, & q = -149 \\ 2^{23}+1 \le c \le 2^{24}-1, & -148 \le q \le 104 \end{cases} \\ \text{double}: &\begin{cases} 1 \le c \le 2^{53}-1, c \ne 2^{52}, & q = -1074 \\ 2^{52}+1 \le c \le 2^{53}-1, & -1073 \le q \le 971 \end{cases} \end{aligned} \tag{6}$$

For irregular floating-point numbers (powers of two), the ranges are:

$$\begin{aligned} \text{float}: &\quad c = 2^{23}, \quad -149 \le q \le 104 \\ \text{double}: &\quad c = 2^{52}, \quad -1074 \le q \le 971 \end{aligned} \tag{7}$$

For subnormal numbers:

$$\begin{aligned} \text{float}: &\quad c \le 2^{23}-1, \quad q = -149 \\ \text{double}: &\quad c \le 2^{52}-1, \quad q = -1074 \end{aligned} \tag{8}$$

The conversion problem can now be formally stated as: given a floating-point value $v = c \cdot 2^q$, find the optimal decimal representation $opt = d \cdot 10^k$ such that:

$$\begin{aligned} &v = c \cdot 2^q \to opt = d \cdot 10^k \\ &\text{subject to: } opt \in R_v, \quad d \in \mathbb{Z}^+, \quad k \in \mathbb{Z} \end{aligned} \tag{9}$$

where R_v is the rounding interval of v as defined in Section 2.

For example, consider the IEEE 754 binary64 floating-point number representing 1.3. Its actual value is 1.3000000000000000444089209850062616169452667236328125, with hexadecimal representation `3ff4ccccccccccd`. The optimal decimal representation opt satisfying the SW principle is simply `1.3`. Similarly, for the binary32 representation of 1.3, with actual value 1.299999952316284179687 5 and hexadecimal representation `3fa66666`, the optimal representation is also `1.3`.

3.3. Review of the Schubfach Algorithm and Our Derivation

This section reviews the Schubfach algorithm and presents our derivation of an optimized variant. We begin by establishing the mathematical foundation for determining the optimal decimal representation.

3.3.1. Candidate Values for the Significand d

The Schubfach algorithm identifies four candidate values for the decimal significand d:

$$d \in \{10\lfloor v \cdot 10^{-k-1} \rfloor, \lfloor 10v \cdot 10^{-k-1} \rfloor, \lfloor 10v \cdot 10^{-k-1} \rfloor + 1, 10\lfloor v \cdot 10^{-k-1} \rfloor + 10\} \tag{10}$$

The exponent k is computed as:

$$k = \begin{cases} \lfloor q \cdot \lg 2 \rfloor & \text{if } v \in \text{regular} \\ \lfloor q \cdot \lg 2 - \lg(4/3) \rfloor & \text{otherwise} \end{cases} \tag{11}$$

 doi:10.20944/preprints202511.1698.v2

For efficient computation on modern processors, Equation (11) can be implemented using integer arithmetic:

$$\begin{aligned} \text{double}: &\quad k = (q \cdot 315653 - (v \in \text{regular} \,?\, 0 : 131072)) \gg 20 \\ \text{float}: &\quad k = (q \cdot 1233 - (v \in \text{regular} \,?\, 0 : 512)) \gg 12 \end{aligned} \tag{12}$$

3.3.2. Decomposition into Integer and Fractional Parts

Let $m = \lfloor v \cdot 10^{-k-1} \rfloor$ denote the integer part and $n = v \cdot 10^{-k-1} - m$ the fractional part, where $0 \leq n < 1$. Substituting into Equation (10):

$$d \in \{10m, \lfloor 10(m+n) \rfloor, \lfloor 10(m+n) \rfloor + 1, 10m + 10\} \tag{13}$$

Since $\lfloor 10(m+n) \rfloor = 10m + \lfloor 10n \rfloor$, the candidates simplify to:

$$d \in \{10m, 10m + \lfloor 10n \rfloor, 10m + \lfloor 10n \rfloor + 1, 10m + 10\} \tag{14}$$

Based on the Schubfach algorithm and the SW principle, if the value $10m$ or $10m + 10$ falls within the range R_v, it is selected as the optimal solution d. In cases where neither $10m$ nor $10m + 10$ lies within R_v, the optimal solution d is determined as either $10m + \lfloor 10n \rfloor$ or $10m + \lfloor 10n \rfloor + 1$ in accordance with the rules of correct rounding. We decompose d as $d = ten + one$, where $ten = 10m$ and $one \in \{0, \lfloor 10n \rfloor, \lfloor 10n \rfloor + 1, 10\}$. The problem thus reduces to determining the appropriate value of one.

$$one = \begin{cases} 0; \text{ if } 10m \in R_v \\ 10; \text{ else if } 10m + 10 \in R_v \\ \lfloor 10n \rfloor \text{ or } \lfloor 10n \rfloor + 1; \text{ else apply correct rounding} \end{cases} \tag{15}$$

3.3.3. Selection Criteria for *One*

The selection of one depends on the relationship between the rounding interval and the candidate values. Recall that the rounding interval for a regular floating-point number $v = c \cdot 2^q$ is $R_v = [v - 2^{q-1}, v + 2^{q-1}]$, where 2^{q-1} represents the half-unit in the last place (ulp) of v.

- **Case $one = 0$ (i.e., $d = 10m$):** This case applies when $10m \cdot 10^k$ is the closest representable decimal to v within the rounding interval. The condition is derived as follows:
  The lower bound of the rounding interval must be less than $10m \cdot 10^k$:

$$\begin{aligned} c \cdot 2^q - 10m \cdot 10^k &< 2^{q-1} \\ c \cdot 2^q - \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor \cdot 10^{k+1} &< 2^{q-1} \\ c \cdot 2^q \cdot 10^{-k-1} - \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor &< 2^{-1} \cdot 2^q \cdot 10^{-k-1} \\ n &< 2^{q-1} \cdot 10^{-k-1} \end{aligned} \tag{16}$$

  When equality holds ($n = 2^{q-1} \cdot 10^{-k-1}$), we apply the round-to-even rule, requiring c to be even:

$$one = 0 \quad \text{if} \quad 2^{q-1} \cdot 10^{-k-1} > n \;\text{ or }\; \left(2^{q-1} \cdot 10^{-k-1} = n \text{ and } c \bmod 2 = 0\right) \tag{17}$$

- **Case $one = 10$ (i.e., $d = 10m + 10$):** This case applies when $(10m + 10) \cdot 10^k$ is the closest representable decimal to v. The condition is derived similarly:
  The upper bound of the rounding interval must be greater than $(10m + 10) \cdot 10^k$:

$$\begin{aligned} (10m + 10) \cdot 10^k - c \cdot 2^q &< 2^{q-1} \\ \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor \cdot 10^{k+1} + 10^{k+1} - c \cdot 2^q &< 2^{q-1} \\ 1 - \left(c \cdot 2^q \cdot 10^{-k-1} - \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor\right) &< 2^{-1} \cdot 2^q \cdot 10^{-k-1} \\ 1 - n &< 2^{q-1} \cdot 10^{-k-1} \end{aligned} \tag{18}$$

 doi:10.20944/preprints202511.1698.v2

When equality holds ($1 - n = 2^{q-1} \cdot 10^{-k-1}$), we again apply round-to-even:

$$one = 10 \quad \text{if} \quad 2^{q-1} \cdot 10^{-k-1} > 1 - n \ \text{ or } \ \left(2^{q-1} \cdot 10^{-k-1} = 1 - n \text{ and } c \bmod 2 = 0\right) \tag{19}$$

- **Case** $one \in \{\lfloor 10n \rfloor, \lfloor 10n \rfloor + 1\}$**:** When neither boundary condition applies, the optimal value lies between $10m + \lfloor 10n \rfloor$ and $10m + \lfloor 10n \rfloor + 1$. We determine one by rounding $10n$ to the nearest integer:
  - If the fractional part $\{10n\} < 0.5$: $one = \lfloor 10n \rfloor$
  - If the fractional part $\{10n\} > 0.5$: $one = \lfloor 10n \rfloor + 1$
  - If the fractional part $\{10n\} = 0.5$: apply round-to-even

  For irregular floating-point numbers (powers of two), additional verification is required to ensure the selected value lies within the rounding interval R_v, as the interval boundaries differ for these special cases.

3.3.4. Algorithm Overview

Algorithm 1 summarizes our optimized variant of the Schubfach algorithm (xjb32 for float, xjb64 for double). Given inputs c and q, the algorithm returns d and k such that $d \cdot 10^k$ satisfies the SW principle. The computation of k follows Equation (12); the remainder of this section focuses on efficient computation of d.

The subsequent sections detail:

1. Lookup table precomputation
2. Efficient computation of m
3. Fast boundary condition testing for $one \in \{0, 10\}$
4. Efficient computation of $\lfloor 10n \rfloor$ and rounding
5. Handling of irregular floating-point numbers
6. implementation of pseudocode

The following will discuss the above content in detail from Section 3.4 to Section 3.9.

3.4. Lookup Table Precomputation

The algorithm in this paper employs a lookup table to store pre-computed values of 10^{-k-1} for different ranges of q: $[-149, 104]$ for float and $[-1074, 971]$ for double. These lookup tables use extended precision: 64-bit for float and 128-bit for double. The reference implementation is available in gen.py.

3.4.1. Fundamental Calculation

Let B denote the bit length of each entry in the lookup table, with $B = 64$ for float and $B = 128$ for double. For any integer e_{10} (representing a power of 10), we aim to represent $10^{e_{10}}$ in the form $f \cdot 2^{\lfloor e_2 \rfloor}$, where $1 \leqslant f < 2$ and e_2 is a real number. This gives:

$$f \cdot 2^{\lfloor e_2 \rfloor} = 2^{e_2} = 10^{e_{10}} \tag{20}$$

Taking the logarithm base 2 of both sides, we get:

$$\lfloor e_2 \rfloor = \lfloor e_{10} \cdot \log_2(10) \rfloor \tag{21}$$

Solving for f gives:

$$f = \frac{10^{e_{10}}}{2^{\lfloor e_{10} \cdot \log_2(10) \rfloor}} \tag{22}$$

The lookup table entries are computed using upward rounding:

 doi:10.20944/preprints202511.1698.v2

Algorithm 1 The xjb Algorithm for Float-to-Decimal Conversion

Require: Floating-point components c (significand) and q (exponent)
Ensure: Decimal representation $d \cdot 10^k$ satisfying the SW principle

$c \cdot 2^q \leftarrow v$
if $v \in$ regular **then**
  $k \leftarrow \lfloor q \cdot \lg 2 \rfloor$
else
  $k \leftarrow \lfloor q \cdot \lg 2 - \lg(4/3) \rfloor$
end if
$m \leftarrow \lfloor v \cdot 10^{-k-1} \rfloor$
$n \leftarrow v \cdot 10^{-k-1} - m$
$ten \leftarrow 10m$
$\delta \leftarrow 10n - \lfloor 10n \rfloor$ {fractional part of $10n$}
if $\delta = 0.5$ **then**
  if $\lfloor 10n \rfloor \bmod 2 = 0$ **then**
    $one \leftarrow \lfloor 10n \rfloor$ {round to even}
  else
    $one \leftarrow \lfloor 10n \rfloor + 1$
  end if
else if $\delta < 0.5$ **then**
  $one \leftarrow \lfloor 10n \rfloor$
else
  $one \leftarrow \lfloor 10n \rfloor + 1$
end if
if $v \in$ irregular **then**
  if $\delta > 2^{q-2} \cdot 10^{-k}$ **then**
    $one \leftarrow \lfloor 10n \rfloor + 1$
  end if
  if $2^{q-2} \cdot 10^{-k-1} \geq n$ **then**
    $one \leftarrow 0$
  end if
else
  if $2^{q-1} \cdot 10^{-k-1} > n$ OR $\left(2^{q-1} \cdot 10^{-k-1} = n \text{ and } c \bmod 2 = 0\right)$ **then**
    $one \leftarrow 0$
  end if
  if $2^{q-1} \cdot 10^{-k-1} > 1 - n$ OR $\left(2^{q-1} \cdot 10^{-k-1} = 1 - n \text{ and } c \bmod 2 = 0\right)$ **then**
    $one \leftarrow 10$
  end if
end if
$d \leftarrow ten + one$

return d, k

 doi:10.20944/preprints202511.1698.v2

$$\begin{aligned} lookup[e_{10}] &= \lceil f \cdot 2^{B-1} \rceil \\ &= \lceil \frac{10^{e_{10}}}{2^{\lfloor e_{10} \cdot \log_2(10) \rfloor}} \cdot 2^{B-1} \rceil \\ &= \lceil 10^{e_{10}} \cdot 2^{B-1-\lfloor e_{10} \cdot \log_2(10) \rfloor} \rceil \end{aligned} \tag{23}$$

Notably, $f \cdot 2^{B-1}$ becomes an integer for certain ranges of e_{10}: $0 \leqslant e_{10} \leqslant 27$ for float and $0 \leqslant e_{10} \leqslant 55$ for double.

3.4.2. Detailed Calculation Process

The detailed calculation process is as follows:

- Float

The range of $-k-1$ is calculated to be [-32, 44] through the q value range in Equation (6), so the lookup table contains representation values from 10 to the power of -32 to 10 to the power of 44. The calculation process is as follows:

$$\begin{aligned} & -32 \leqslant e_{10} \leqslant 44 \\ & e_2 = |\lfloor e_{10} \cdot \log_2(10) \rfloor - 63| \\ & pow10t = \begin{cases} 2^{e_2} // 10^{|e10|}; \text{if } e_{10} < 0 \\ 10^{|e10|} // 2^{e_2}; \text{if } e_{10} \geqslant 20 \\ 10^{|e10|} \cdot 2^{e_2}; \text{if } 1 \leqslant e_{10} \leqslant 19 \end{cases} \\ & f_{1,e_{10}} = pow10 = pow10t + (e_{10} \geqslant 0 \&\& e_{10} \leqslant 27 ? 0 : 1) \end{aligned} \tag{24}$$

When $0 \leqslant e_{10} \leqslant 27$, the lookup table variable indicates that the values $f_{1,e_{10}} \cdot 2^{\lfloor e_{10} \cdot \log_2(10) \rfloor - 63}$ and $10^{e_{10}}$ are equal. In other cases, the relative error is less than 2^{-63}. Expressed as:

$$\begin{aligned} r_{1,e_{10}} &= \frac{f_{1,e_{10}} \cdot 2^{\lfloor e_{10} \cdot \log_2(10) \rfloor - 63}}{10^{e_{10}}} \\ &\in \begin{cases} 1; \text{if } 0 \leqslant e_{10} \leqslant 27 \\ (1, 1 + 2^{-63}); \text{if } e_{10} < 0 \text{ or } e_{10} > 27 \end{cases} \end{aligned} \tag{25}$$

- Double

The range of $-k-1$ is calculated to be [-293, 323] through the q value range in Equation (6), so the lookup table contains representation values from 10 to the power of -293 to 10 to the power of 323. The calculation process is as follows:

$$\begin{aligned} & -293 \leqslant e_{10} \leqslant 323 \\ & e_2 = |\lfloor e_{10} \cdot \log_2(10) \rfloor - 127| \\ & pow10t = \begin{cases} 2^{e_2} // 10^{|e10|}; \text{if } e_{10} < 0 \\ 10^{|e10|} // 2^{e_2}; \text{if } e_{10} \geqslant 39 \\ 10^{|e10|} \cdot 2^{e_2}; \text{if } 1 \leqslant e_{10} \leqslant 38 \end{cases} \\ & f_{1,e_{10}} = pow10 = pow10t + (e_{10} \geqslant 0 \&\& e_{10} \leqslant 55 ? 0 : 1) \end{aligned} \tag{26}$$

When $0 \leqslant e_{10} \leqslant 55$, the lookup table variable indicates that the values $f_{1,e_{10}} \cdot 2^{\lfloor e_{10} \cdot \log_2(10) \rfloor - 127}$ and $10^{e_{10}}$ are equal. In other cases, the relative error is less than 2^{-127}. Expressed as:

$$\begin{aligned} r_{1,e_{10}} &= \frac{f_{1,e_{10}} \cdot 2^{\lfloor e_{10} \cdot \log_2(10) \rfloor - 127}}{10^{e_{10}}} \\ &\in \begin{cases} 1; \text{if } 0 \leqslant e_{10} \leqslant 55 \\ (1, 1 + 2^{-127}); \text{if } e_{10} < 0 \text{ or } e_{10} > 55 \end{cases} \end{aligned} \tag{27}$$

 doi:10.20944/preprints202511.1698.v2

Let r_1 denote the error for float lookup table entries, r_2 for double entries, and r for both. In Algorithm (1), we retrieve 10^{-k-1} from the lookup table. The lookup table provides exact values when q falls within specific ranges:

$$\begin{aligned} \text{float} &: 0 \leqslant -k-1 \leqslant 27 \Rightarrow -93 \leqslant q \leqslant -1 \\ \text{double} &: 0 \leqslant -k-1 \leqslant 55 \Rightarrow -186 \leqslant q \leqslant -1 \end{aligned} \tag{28}$$

For q outside these ranges, the lookup table entries have bounded relative errors:

$$\begin{aligned} \text{float} &: 0 < r_1 - 1 < 2^{-63} \\ \text{double} &: 0 < r_2 - 1 < 2^{-127} \end{aligned} \tag{29}$$

3.4.3. Storage Requirements

The float lookup table requires 616 bytes of storage, calculated as $(44 - (-32) + 1) \times 8$ bytes (77 entries × 8 bytes each). The double lookup table requires 9872 bytes, calculated as $(323 - (-293) + 1) \times 16$ bytes (617 entries × 16 bytes each).

3.4.4. Implementation Notes

The lookup table pre-computation uses efficient integer arithmetic to avoid precision loss during calculations. The conditional logic in Equations (24) and (26) optimizes the computation based on the sign and magnitude of e_{10}, ensuring efficient generation of accurate lookup table entries.

3.5. Efficient Computation of m

This section presents an efficient method for calculating m in Algorithm (1), which is defined as $m = \lfloor v \cdot 10^{-k-1} \rfloor$.

3.5.1. Relevant Theorems

We start with some theorems from the Dragonbox algorithm paper:

Let n, P, and Q be positive integers where P and Q are coprime, $P < Q$, $1 \leqslant n \leqslant n_{max}$, and $Q > n_{max}$. Let P^*/Q^* be the best rational approximation of P/Q with $Q^* \leqslant n_{max}$ and P_*/Q_* be the best rational approximation with $Q_* \leqslant n_{max}$.

If $n \cdot P$ does not divide Q evenly:

$$\lfloor n \cdot \frac{P}{Q} \rfloor + 1 = \lceil n \cdot \frac{P}{Q} \rceil \tag{30}$$

Suppose the following holds:

$$\lfloor n \cdot \frac{P}{Q} \rfloor = \lfloor n \cdot \xi \rfloor \tag{31}$$

Then:

$$\frac{P_*}{Q_*} = \max_{1 \leqslant n \leqslant n_{max}} \frac{\lfloor n \cdot \frac{P}{Q} \rfloor}{n} \leqslant \xi < \min_{1 \leqslant n \leqslant n_{max}} \frac{\lfloor n \cdot \frac{P}{Q} \rfloor + 1}{n} = \min_{1 \leqslant n \leqslant n_{max}} \frac{\lceil n \cdot \frac{P}{Q} \rceil}{n} = \frac{P^*}{Q^*} \tag{32}$$

Thus, the range of ξ is:

$$\frac{P_*}{Q_*} \leqslant \xi < \frac{P^*}{Q^*} \tag{33}$$

The range of the fractional part of $n \cdot \frac{P}{Q}$ is:

$$\left[\frac{(Q_* P) \% Q}{Q}, \frac{(Q^* P) \% Q}{Q} \right] \tag{34}$$

Here, the fractional part is minimized when $n = Q_*$ and maximized when $n = Q^*$.

 doi:10.20944/preprints202511.1698.v2

3.5.2. Best Rational Approximation Function

We define the best rational approximation function (implemented in test1.py):

$$(\frac{P_*}{Q_*}, \frac{P^*}{Q^*}) = (DN, UP) = f(C, P, Q) \tag{35}$$

This function $f(C, P, Q)$ calculates the best rational approximation with denominator not exceeding C using the Farey sequence median theorem. DN and UP are adjacent terms in the C-th order Farey sequence F_C. Therefore, we can calculate Equation (34) very quickly.

3.5.3. Key Proof

In Algorithm (1), we need to compute $m = \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor$. We aim to prove:

$$m = \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor c \cdot 2^q \cdot r \cdot 10^{-k-1} \rfloor \tag{36}$$

Where r is the lookup table error defined in Equations (25) and (27). When condition (28) is met, $r = 1$, and the equation holds trivially. For $r \neq 1$:

$$\begin{aligned} \text{float} &: 1 < r < 1 + 2^{-63} \\ \text{double} &: 1 < r < 1 + 2^{-127} \end{aligned} \tag{37}$$

Calculate the range of $2^q \cdot 10^{-k-1}$ and we get:

$$2^q \cdot 10^{-k-1} = 10^{-1} \cdot \left(2^q \cdot 10^{-\lfloor q \cdot \lg(2) \rfloor}\right) = 10^{-1} \cdot \left(10^{q \cdot \lg(2) - \lfloor q \cdot \lg(2) \rfloor}\right) \tag{38}$$

When q is not 0, Equation (38) exists:

$$\begin{aligned} q \cdot \lg(2) &\neq \lfloor q \cdot \lg(2) \rfloor \\ 0 < q \cdot \lg(2) &- \lfloor q \cdot \lg(2) \rfloor < 1 \end{aligned} \tag{39}$$

When q is 0, $q \cdot \lg(2) - \lfloor q \cdot \lg(2) \rfloor = 0$, so the final conclusion is:

$$10^{-1} \leqslant 2^q \cdot 10^{-k-1} < 1 \tag{40}$$

Because there is:

$$c \cdot 2^q \cdot 10^{-k-1} = c \cdot \frac{2^{q-k-1}}{5^{k+1}} \in [0.1c, c) \tag{41}$$

Therefore:

$$c \cdot 2^q \cdot 10^{-k-1} = \begin{cases} \frac{c \cdot 2^{q-k-1}}{5^{k+1}}; q \geqslant 1 \\ \frac{c}{2^{1+k-q} \cdot 5^{k+1}} = \frac{c}{10}; q = 0 \\ \frac{c \cdot 5^{-k-1}}{2^{1+k-q}}; q < 0 \end{cases} \tag{42}$$

Suppose:

$$c \cdot 2^q \cdot 10^{-k-1} = c \cdot \frac{x}{y} < c \tag{43}$$

Then there are:

$$(x, y) = \begin{cases} \left(2^{q-k-1}, 5^{k+1}\right); q \geqslant 1 \\ (1, 10); q = 0 \\ \left(5^{-k-1}, 2^{1+k-q}\right); q < 0 \end{cases} \tag{44}$$

3.5.4. Bit Width Calculation

Define maximum values for c:

 doi:10.20944/preprints202511.1698.v2

$$\begin{aligned} \text{float} : c \leqslant c_{max} = C_1 = 2^{24} - 1 \\ \text{double} : c \leqslant c_{max} = C_2 = 2^{53} - 1 \end{aligned} \tag{45}$$

Let C denote either C_1 or C_2 depending on the precision.
For $y > C$, compute P^* and Q^* for each q using $f(C, x, y)$ and find the minimum BIT such that:

$$\frac{x}{y}(1 + 2^{-BIT}) < \frac{P^*}{Q^*} \tag{46}$$

For $y \leqslant C$:

$$c \cdot \frac{x}{y}\left(1 + \frac{1}{Cy}\right) = \frac{cx + \frac{c}{C} \cdot \frac{x}{y}}{y} < \frac{cx + 1}{y} \tag{47}$$

Thus:

$$\lfloor c \cdot \frac{x}{y} \rfloor = \lfloor c \cdot \frac{x}{y}\left(1 + \frac{1}{Cy}\right) \rfloor \tag{48}$$

Similarly, find the minimum BIT such that:

$$\frac{x}{y}(1 + 2^{-BIT}) < \frac{x}{y}\left(1 + \frac{1}{Cy}\right) \tag{49}$$

3.5.5. Results

The maximum of the minimum BIT values for all q (calculated in test1.py in about 1-2 seconds) is:

$$\begin{aligned} \text{float} : BIT_{max} = 52 \\ \text{double} : BIT_{max} = 113 \end{aligned} \tag{50}$$

Thus:

$$\begin{aligned} \text{float} : \lfloor c \cdot \frac{x}{y} \rfloor = \lfloor c \cdot \frac{x}{y} \cdot (1 + 2^{-52}) \rfloor = \lfloor c \cdot \frac{x}{y} \cdot r_1 \rfloor \\ \text{double} : \lfloor c \cdot \frac{x}{y} \rfloor = \lfloor c \cdot \frac{x}{y} \cdot (1 + 2^{-113}) \rfloor = \lfloor c \cdot \frac{x}{y} \cdot r_2 \rfloor \end{aligned} \tag{51}$$

This result confirms that m can be calculated efficiently using the lookup table values, even with their inherent errors. Once m is determined, $ten = 10m$ can be computed quickly.

3.6. Fast Boundary Condition Testing for $one = 0$ and $one = 10$

In Algorithm 1, the conditions for determining $one = 0$ and $one = 10$ appear on lines 31 and 34, respectively. This section introduces an optimized method to quickly test these boundary conditions using equivalent mathematical formulations.

3.6.1. Equivalent Conditions for Boundary Testing

We start by deriving equivalent mathematical conditions for testing $one = 0$ and $one = 10$.

Case 1: Testing $one = 0$

When $2^{-1} \cdot 2^q \cdot 10^{-k-1} = n$, this is equivalent to:

$$\begin{aligned} c \cdot 2^q \cdot 10^{-k-1} - \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor = 2^{-1} \cdot 2^q \cdot 10^{-k-1} \\ (2c - 1) \cdot 2^{q-1} \cdot 10^{-k-1} = \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor \end{aligned} \tag{52}$$

 doi:10.20944/preprints202511.1698.v2

Case 2: Testing $one = 10$

When $2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n$, this is equivalent to:

$$\begin{aligned} \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor - c \cdot 2^q \cdot 10^{-k-1} + 1 &= 2^{-1} \cdot 2^q \cdot 10^{-k-1} \\ (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} &= \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor + 1 \end{aligned} \tag{53}$$

3.6.2. Integer Testing Analysis

To further analyze these conditions, we start with the range of $2^{q-1} \cdot 10^{-k-1}$ from Equation (40):

$$2^{q-1} \cdot 10^{-k-1} \in [0.05, 0.5) \tag{54}$$

Analysis for $one = 0$

For the $one = 0$ case, we can derive:

$$\begin{aligned} \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor - 1 &< c \cdot 2^q \cdot 10^{-k-1} - 0.5 \\ &< (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} \\ &\leqslant c \cdot 2^q \cdot 10^{-k-1} - 0.05 < \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor + 1 \end{aligned} \tag{55}$$

This implies that when $(2c-1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is an integer, it must equal $\lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor$.

Analysis for $one = 10$

Similarly, for the $one = 10$ case:

$$\begin{aligned} \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor &< c \cdot 2^q \cdot 10^{-k-1} + 0.05 \\ &\leqslant (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} \\ &< c \cdot 2^q \cdot 10^{-k-1} + 0.5 < \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor + 2 \end{aligned} \tag{56}$$

This implies that when $(2c+1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is an integer, it must equal $\lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor + 1$.

3.6.3. Key Insight: Integer Divisibility Test

The key insight is that testing for $one = 0$ or $one = 10$ is equivalent to checking whether $(2c \pm 1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is an integer. This can be rewritten as:

$$(2c \pm 1) \cdot 2^{q-1} \cdot 10^{-k-1} = (2c \pm 1) \cdot 2^{q-k-2} \cdot 5^{-k-1} \tag{57}$$

We analyze different ranges of q to simplify this condition:

- **Case $q \geqslant 2$:** From $q \geqslant 2$, we get $k \geqslant 0$. The expression simplifies to checking if $(2c \pm 1) \cdot 2^{q-k-2}$ is divisible by 5^{k+1}. Since 2 and 5 are coprime, this reduces to checking if $(2c \pm 1)$ is divisible by 5^{k+1}:

$$(2c \pm 1) \mod 5^{k+1} = 0 \tag{58}$$

Let t be a positive integer such that $2c \pm 1 = t \cdot 5^{k+1}$. Since $2c \pm 1$ is odd, t must also be odd. Considering the ranges of c for float and double:

$$\begin{aligned} \text{float} &: 2c-1 \in [2^{24}+1, 2^{25}-3];\ 2c+1 \in [2^{24}+3, 2^{25}-1]; \\ \text{double} &: 2c-1 \in [2^{53}+1, 2^{54}-3];\ 2c+1 \in [2^{53}+3, 2^{54}-1]; \end{aligned} \tag{59}$$

 doi:10.20944/preprints202511.1698.v2

This gives us the range for t:

$$\begin{aligned} \text{float}: \frac{2^{24}+1}{5^{k+1}} \leqslant t \leqslant \frac{2^{25}-1}{5^{k+1}}; \\ \text{double}: \frac{2^{53}+1}{5^{k+1}} \leqslant t \leqslant \frac{2^{54}-1}{5^{k+1}}; \end{aligned} \tag{60}$$

The maximum values of k where t can be at least one odd integer are:

$$\begin{aligned} \text{float}: k_{\max} = 9 \Rightarrow q_{\max} = 33, t = 3 \\ \text{double}: k_{\max} = 22 \Rightarrow q_{\max} = 76, t = 1 \end{aligned} \tag{61}$$

- **Case** $1 \geqslant q \geqslant 0$**:** The denominator $2^{2+k-q} \cdot 5^{k+1}$ is even, while the numerator $(2c \pm 1)$ is odd, so no solution exists.
- **Case** $q < 0$**:** The denominator 2^{2+k-q} is even, while the numerator $(2c \pm 1) \cdot 5^{-k-1}$ is odd, so no solution exists.

3.6.4. Summary of Boundary Conditions

In summary, the situations when $(2c \pm 1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is an integer are as follows:

$$\begin{aligned} \text{float}: 2 \leqslant q \leqslant 33 \,\&\&\, (2c \pm 1) \mod 5^{k+1} = 0; \\ \text{double}: 2 \leqslant q \leqslant 76 \,\&\&\, (2c \pm 1) \mod 5^{k+1} = 0; \end{aligned} \tag{62}$$

The range of $-k-1$ is:

$$\begin{aligned} \text{float}: -10 \leqslant -k-1 \leqslant -1 \\ \text{double}: -23 \leqslant -k-1 \leqslant -1 \end{aligned} \tag{63}$$

3.6.5. Efficient Implementation

We can further simplify the testing conditions using bitwise operations. For $one = 0$:

$$\begin{aligned} \text{float}: \lfloor 2^{35} \cdot 2^{q} \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor \\ \text{double}: \lfloor 2^{63} \cdot 2^{q} \cdot 10^{-k-1} \rfloor = \lfloor 2^{64} \cdot n \rfloor \end{aligned} \tag{64}$$

When $2^{-1} \cdot 2^{q} \cdot 10^{-k-1} = 1 - n$, the following conclusions can be drawn:

$$\begin{aligned} \text{float}: \begin{cases} 2^{35} \cdot 2^{q} \cdot 10^{-k-1} = 2^{36} - 2^{36} \cdot n \Rightarrow \\ \lfloor 2^{35} \cdot 2^{q} \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} - 2^{36} \cdot n \rfloor = 2^{36} - 1 - \lfloor 2^{36} \cdot n \rfloor \end{cases} \\ \text{double}: \begin{cases} 2^{63} \cdot 2^{q} \cdot 10^{-k-1} = 2^{64} - 2^{64} \cdot n \Rightarrow \\ \lfloor 2^{63} \cdot 2^{q} \cdot 10^{-k-1} \rfloor = \lfloor 2^{64} - 2^{64} \cdot n \rfloor = 2^{64} - 1 - \lfloor 2^{64} \cdot n \rfloor \end{cases} \end{aligned} \tag{65}$$

The discussion on whether $\lfloor 2^{36} - 2^{36} \cdot n \rfloor = 2^{36} - 1 - \lfloor 2^{36} \cdot n \rfloor$ in Equation (65) holds true, that is, whether $2^{36} \cdot n$ in Equation (65) is an integer, or equivalent to discussing whether the following values are integers when Equation (62) holds true (the same applies to double) :

$$\begin{aligned} \text{float}: 2^{36} \cdot (m+n) = c \cdot 2^{q+36} \cdot 10^{-k-1} = c \cdot 2^{q-k+35} \cdot 5^{-k-1} = c \cdot \frac{2^{q-k+35}}{5^{k+1}} \\ \text{double}: 2^{64} \cdot (m+n) = c \cdot 2^{q+64} \cdot 10^{-k-1} = c \cdot 2^{q-k+63} \cdot 5^{-k-1} = c \cdot \frac{2^{q-k+63}}{5^{k+1}} \end{aligned} \tag{66}$$

Suppose c can divide 5^{k+1} evenly (where t is a temporary integer variable):

$$c = t \cdot 5^{k+1}; t \geqslant 1 \tag{67}$$

 doi:10.20944/preprints202511.1698.v2

Therefore, when Equation (67) was established, there were:

$$2c \pm 1 = 2 \cdot t \cdot 5^{k+1} \pm 1 \tag{68}$$

Expression (68) cannot divide 5^{k+1} evenly, which contradicts Equation (62), so c cannot divide 5^{k+1} evenly. Therefore, for float, $c \cdot 2^{q+36} \cdot 10^{-k-1}$ and $2^{36} \cdot n$ are not integers; For double, $c \cdot 2^{64+q} \cdot 10^{-k-1}$ and $2^{64} \cdot n$ are not integers, that is:

$$\begin{aligned} \text{float}: &\lfloor 2^{36} - 2^{36} \cdot n \rfloor = 2^{36} + \lfloor -2^{36} \cdot n \rfloor = 2^{36} - 1 - \lfloor 2^{36} \cdot n \rfloor \\ \text{double}: &\lfloor 2^{64} - 2^{64} \cdot n \rfloor = 2^{64} + \lfloor -2^{64} \cdot n \rfloor = 2^{64} - 1 - \lfloor 2^{64} \cdot n \rfloor \end{aligned} \tag{69}$$

Therefore, the conclusion (65) is correct. Discuss the necessary and sufficient conditions for whether $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor$ is $2^{-1} \cdot 2^q \cdot 10^{-k-1} = n$. The same applies to double, expressed as:

$$\begin{aligned} \text{float}: &2^{-1} \cdot 2^q \cdot 10^{-k-1} = n \Leftrightarrow \lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor \\ \text{double}: &2^{-1} \cdot 2^q \cdot 10^{-k-1} = n \Leftrightarrow \lfloor 2^{63} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{64} \cdot n \rfloor \end{aligned} \tag{70}$$

Similarly, the necessary and sufficient conditions for whether $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} - 2^{36} \cdot n \rfloor$ is $2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n$. The same applies to double, expressed as:

$$\begin{aligned} \text{float}: &2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n \Leftrightarrow \lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} - 2^{36} \cdot n \rfloor \\ \text{double}: &2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n \Leftrightarrow \lfloor 2^{63} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{64} - 2^{64} \cdot n \rfloor \end{aligned} \tag{71}$$

The sufficient conditions of Equations (70) and (71) are obviously established. Introduce the proof that Equation (70) holds. For float, only the necessary conditions need to be discussed, that is, whether $2^{-1} \cdot 2^q \cdot 10^{-k-1} = n$ must hold true when $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor$ holds, or equivalent to $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor \neq \lfloor 2^{36} \cdot n \rfloor$ must hold true when $2^{-1} \cdot 2^q \cdot 10^{-k-1} \neq n$. The following is proved by proof by contradiction.

Assume that $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor$ holds when $2^{-1} \cdot 2^q \cdot 10^{-k-1} \neq n$. Then there is:

$$\begin{aligned} &\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor \\ \Rightarrow\ &0 < \left| 2^{35} \cdot 2^q \cdot 10^{-k-1} - 2^{36} \cdot n \right| < 1 \\ \Rightarrow\ &0 < \left| (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} - m \right| < 2^{-36} \end{aligned} \tag{72}$$

As is known from Equation (55), there is:

$$m - 1 < (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} < m + 1 \tag{73}$$

Suppose the decimal part of $(2c-1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is represented as n^-, thus we have:

$$\left| (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} - m \right| = \begin{cases} n^-; \text{if } (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} > m \\ 1 - n^-; \text{if } (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} < m \end{cases} \tag{74}$$

Substitute expression (74) into expression (72), and we get:

$$\begin{aligned} &0 < \left| (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} - m \right| < 2^{-36} \\ \Rightarrow\ &0 < n^- < 2^{-36} \text{ or } 0 < 1 - n^- < 2^{-36} \end{aligned} \tag{75}$$

 doi:10.20944/preprints202511.1698.v2

Similarly, it can be known that the double range is the range of n^-. Therefore, there is:

$$\begin{aligned} \text{float}: & n^- \in \left(0, 2^{-36}\right) \cup \left(1-2^{-36}, 1\right) \\ \text{double}: & n^- \in \left(0, 2^{-64}\right) \cup \left(1-2^{-64}, 1\right) \end{aligned} \tag{76}$$

When $2^{-1} \cdot 2^q \cdot 10^{-k-1} \neq n$, it is known from Equation (52) that $(2c-1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is not an integer. Therefore, there is:

$$0 < n^- < 1 \tag{77}$$

It is only necessary to prove that Equation (76) does not hold. Discuss the range of the decimal part n^- when $(2c-1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is not an integer. According to Equation (57), there are:

$$(2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} = (2c-1) \cdot \frac{x}{y} = \begin{cases} \frac{(2c-1) \cdot 2^{q-k-2}}{5^{k+1}}; q \geqslant 2 \\ \frac{(2c-1)}{2^{2+k-q} \cdot 5^{k+1}}; 1 \geqslant q \geqslant 0 \\ \frac{(2c-1) \cdot 5^{-k-1}}{2^{2+k-q}}; q < 0 \end{cases} \tag{78}$$

The maximum value of $2c-1$ is:

$$\begin{aligned} \text{float}: & (2c-1)_{\max} = 2^{25} - 3 \\ \text{double}: & (2c-1)_{\max} = 2^{54} - 3 \end{aligned} \tag{79}$$

Discuss based on the denominator range in Equation (78).

- $y \leqslant (2c-1)_{\max}$
  When $y \leqslant (2c-1)_{\max}$, $y_{\max}$ is the expression (79), the following holds true:

$$\begin{aligned} \frac{1}{y_{\max}} \leqslant n^- \leqslant 1 - \frac{1}{y_{\max}} \\ \frac{1}{y_{\max}} \leqslant 1 - n^- \leqslant 1 - \frac{1}{y_{\max}} \end{aligned} \tag{80}$$

  Therefore, when $y \leqslant (2c-1)_{\max}$, Equation (76) does not hold true.
- $y > (2c-1)_{\max}$
  Call function (35) to calculate the approximation results $P_* \Big/ Q_*$ and $P^* \Big/ Q^*$ of all possible upper and lower limit rational numbers:

$$\left(\frac{P_*}{Q_*}, \frac{P^*}{Q^*}\right) = f((2c-1)_{\max}, x, y) \tag{81}$$

  Therefore, for n^-, the following conclusion can be drawn from formula (34).

$$n^- \in \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \tag{82}$$

  By exhausting all possibilities, we thus have (the test code file is test3.py) :

$$\begin{aligned} \text{float}: & 2^{-33} < n^- < 1 - 2^{-29} \\ \text{double}: & 2^{-62} < n^- < 1 - 2^{-63} \end{aligned} \tag{83}$$

 doi:10.20944/preprints202511.1698.v2

$$\begin{aligned}
\text{float}: &\left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \cap \left(0, 2^{-36}\right) = \varnothing \\
&\left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \cap \left(1 - 2^{-36}, 1\right) = \varnothing \\
\text{double}: &\left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \cap \left(0, 2^{-64}\right) = \varnothing \\
&\left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \cap \left(1 - 2^{-64}, 1\right) = \varnothing
\end{aligned} \tag{84}$$

Therefore, when $y > (2c-1)_{\max}$, Equation (76) does not hold true.

In summary, when $2^{-1} \cdot 2^q \cdot 10^{-k-1} \neq n$, Equation (76) does not hold true, that is, $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor \neq \lfloor 2^{36} \cdot n \rfloor$ must hold true. Therefore, when $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor$ holds, $2^{-1} \cdot 2^q \cdot 10^{-k-1} = n$ must hold true. Therefore, Equation (70) holds.

Similarly, it can be proved that when $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} - 2^{36} \cdot n \rfloor$ holds, $2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n$ must hold true. The same applies to double. Similarly, by proof of contradiction, for float, it is assumed that when $2^{-1} \cdot 2^q \cdot 10^{-k-1} \neq 1 - n$ holds, $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} - 2^{36} \cdot n \rfloor$ holds. That is:

$$\begin{aligned}
&\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} - 2^{36} \cdot n \rfloor \\
&\Rightarrow 0 < \left| 2^{35} \cdot 2^q \cdot 10^{-k-1} - 2^{36} + 2^{36} \cdot n \right| < 1 \\
&\Rightarrow 0 < \left| 2^{q-1} \cdot 10^{-k-1} - 1 + n \right| < 2^{-36} \\
&\Rightarrow -2^{-36} < (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} - m - 1 < 2^{-36}
\end{aligned} \tag{85}$$

As is known from Equation (56), there is:

$$m < (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} < m + 2 \tag{86}$$

Suppose the decimal part of $(2c+1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is represented as n^+, thus we have:

$$(2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} - m - 1 = \begin{cases} n^+; \text{if } (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} > m+1 \\ 1 - n^+; \text{if } (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} < m+1 \end{cases} \tag{87}$$

Substitute expression (87) into expression (85), and we get:

$$\begin{aligned}
&0 < \left| (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} - m - 1 \right| < 2^{-36} \\
&\Rightarrow 0 < 1 - n^+ < 2^{-36} \text{ or } 0 < n^+ < 2^{-36}
\end{aligned} \tag{88}$$

Similarly, it can be known that the double range is the range of n^+. Therefore, there is:

$$\begin{aligned}
\text{float}: n^+ &\in \left(0, 2^{-36}\right) \cup \left(1 - 2^{-36}, 1\right) \\
\text{double}: n^+ &\in \left(0, 2^{-64}\right) \cup \left(1 - 2^{-64}, 1\right)
\end{aligned} \tag{89}$$

When $2^{-1} \cdot 2^q \cdot 10^{-k-1} \neq 1 - n$, it is known from Equation (53) that $(2c+1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is not an integer. Therefore, there is:

$$0 < n^+ < 1 \tag{90}$$

 doi:10.20944/preprints202511.1698.v2

It is only necessary to prove that Equation (89) does not hold. Discuss the range of the decimal part n^+ when $(2c+1)\cdot 2^{q-1}\cdot 10^{-k-1}$ is not an integer. According to Equation (57), there are:

$$(2c+1)\cdot 2^{q-1}\cdot 10^{-k-1} = (2c+1)\cdot\frac{x}{y} = \begin{cases} \frac{(2c+1)\cdot 2^{q-k-2}}{5^{k+1}}; q \geqslant 2 \\ \frac{(2c+1)}{2^{2+k-q}\cdot 5^{k+1}}; 1 \geqslant q \geqslant 0 \\ \frac{(2c+1)\cdot 5^{-k-1}}{2^{2+k-q}}; q < 0 \end{cases} \tag{91}$$

The maximum value of $2c+1$ is:

$$\begin{aligned} \text{float}: (2c+1)_{\max} &= 2^{25}-1 \\ \text{double}: (2c+1)_{\max} &= 2^{54}-1 \end{aligned} \tag{92}$$

Discuss based on the denominator range in Equation (91).

- $y \leqslant (2c+1)_{\max}$
  When $y \leqslant (2c+1)_{\max}$, $y_{\max}$ is the expression (92), the following holds true:

$$\begin{aligned} \frac{1}{y_{\max}} \leqslant n^+ &\leqslant 1-\frac{1}{y_{\max}} \\ \frac{1}{y_{\max}} \leqslant 1-n^+ &\leqslant 1-\frac{1}{y_{\max}} \end{aligned} \tag{93}$$

  Therefore, when $y \leqslant (2c+1)_{\max}$, Equation (89) does not hold true.
- $y > (2c+1)_{\max}$
  Call function (35) to calculate the approximation results $\frac{P_*}{Q_*}$ and $\frac{P^*}{Q^*}$ of all possible upper and lower limit rational numbers:

$$\left(\frac{P_*}{Q_*}, \frac{P^*}{Q^*}\right) = f((2c+1)_{\max}, x, y) \tag{94}$$

  Therefore, for n^+, the following conclusion can be drawn from formula (34).

$$n^+ \in \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \tag{95}$$

  By exhausting all possibilities, we thus have (the test code file is test7.py) :

$$\begin{aligned} \text{float}: 2^{-33} < n^+ &< 1-2^{-29} \\ \text{double}: 2^{-62} < n^+ &< 1-2^{-63} \end{aligned} \tag{96}$$

$$\begin{aligned} \text{float}: \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \cap \left(0, 2^{-36}\right) &= \varnothing \\ \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \cap \left(1-2^{-36}, 1\right) &= \varnothing \\ \text{double}: \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \cap \left(0, 2^{-64}\right) &= \varnothing \\ \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \cap \left(1-2^{-64}, 1\right) &= \varnothing \end{aligned} \tag{97}$$

  Therefore, when $y > (2c+1)_{\max}$, Equation (89) does not hold true.

In summary, when $2^{-1}\cdot 2^q\cdot 10^{-k-1} \neq 1-n$, Equation (89) does not hold true, that is, $\lfloor 2^{35}\cdot 2^q\cdot 10^{-k-1}\rfloor \neq \lfloor 2^{36}-2^{36}\cdot n\rfloor$ must hold true. Therefore, when $\lfloor 2^{35}\cdot 2^q\cdot 10^{-k-1}\rfloor = \lfloor 2^{36}-2^{36}\cdot n\rfloor$ holds, $2^{-1}\cdot 2^q\cdot 10^{-k-1} = 1-n$ must hold true. The same is true for double.Therefore, Equation (71) holds.

 doi:10.20944/preprints202511.1698.v2

The following conclusions hold:

$$\begin{aligned} \text{float}: \lfloor 2^{36} - 2^{36} \cdot n \rfloor &= \begin{cases} 2^{36} - 1 - \lfloor 2^{36} \cdot n \rfloor; \text{if } c \cdot 2^{36+q} \cdot 10^{-k-1} \notin Z \\ 2^{36} - \lfloor 2^{36} \cdot n \rfloor; \text{if } c \cdot 2^{36+q} \cdot 10^{-k-1} \in Z \end{cases} \\ \text{double}: \lfloor 2^{64} - 2^{64} \cdot n \rfloor &= \begin{cases} 2^{64} - 1 - \lfloor 2^{64} \cdot n \rfloor; \text{if } c \cdot 2^{64+q} \cdot 10^{-k-1} \notin Z \\ 2^{64} - \lfloor 2^{64} \cdot n \rfloor; \text{if } c \cdot 2^{64+q} \cdot 10^{-k-1} \in Z \end{cases} \end{aligned} \tag{98}$$

Discuss whether the following Equation (99) holds when conditions (62) and (63) are met:

$$\begin{aligned} \text{float}: \lfloor c \cdot \frac{2^{q+35-k}}{5^{k+1}} \rfloor &= \lfloor c \cdot \frac{2^{q+35-k}}{5^{k+1}} \cdot r \rfloor \\ &= \lfloor c \cdot \frac{2^{q+35-k}}{5^{k+1}} \cdot \frac{\left(2^{63-\lfloor (-k-1) \cdot \log_2(10) \rfloor} // 10^{k+1}\right) + 1}{10^{-k-1}} \cdot 2^{\lfloor (-k-1) \cdot \log_2(10) \rfloor - 63} \rfloor \\ \text{double}: \lfloor c \cdot \frac{2^{q+63-k}}{5^{k+1}} \rfloor &= \lfloor c \cdot \frac{2^{q+63-k}}{5^{k+1}} \cdot r \rfloor \\ &= \lfloor c \cdot \frac{2^{q+63-k}}{5^{k+1}} \cdot \frac{\left(2^{127-\lfloor (-k-1) \cdot \log_2(10) \rfloor} // 10^{k+1}\right) + 1}{10^{-k-1}} \cdot 2^{\lfloor (-k-1) \cdot \log_2(10) \rfloor - 127} \rfloor \end{aligned} \tag{99}$$

There are:

$$\begin{aligned} \text{float}: \lfloor c \cdot \frac{2^{q+35-k}}{5^{k+1}} \rfloor &= \lfloor 2^{36} \cdot (m+n) \rfloor = 2^{36} \cdot m + \lfloor 2^{36} \cdot n \rfloor \\ \text{double}: \lfloor c \cdot \frac{2^{q+63-k}}{5^{k+1}} \rfloor &= \lfloor 2^{64} \cdot (m+n) \rfloor = 2^{64} \cdot m + \lfloor 2^{64} \cdot n \rfloor \end{aligned} \tag{100}$$

It has been proven earlier that m can be accurately calculated. Then, when (99) holds true, the values $\lfloor 2^{36} \cdot n \rfloor$ and $\lfloor 2^{64} \cdot n \rfloor$ on the right side of Equations (64) and (65) can be accurately calculated.

From Equation (57), we have:

$$c = \frac{t \cdot 5^{k+1} - 1}{2} \tag{101}$$

Substituting Equation (101) into Equation (99), we have:

$$\begin{aligned} \text{float}: c \cdot \frac{2^{q+35-k}}{5^{k+1}} &= t \cdot 2^{q+34-k} - \frac{2^{q+34-k}}{5^{k+1}} \\ \text{double}: c \cdot \frac{2^{q+63-k}}{5^{k+1}} &= t \cdot 2^{q+62-k} - \frac{2^{q+62-k}}{5^{k+1}} \end{aligned} \tag{102}$$

When conditions (62) and (63) are met, $t \cdot 2^{q+34-k}$ and $t \cdot 2^{q+62-k}$ are integers. Under the condition of meeting condition (62), the decimal part of expression (102) is represented as:

$$\begin{aligned} \text{float}: \frac{2^{q+34-k} \% 5^{k+1}}{5^{k+1}}; 2 \leqslant q \leqslant 33 \\ \text{double}: \frac{2^{q+62-k} \% 5^{k+1}}{5^{k+1}}; 2 \leqslant q \leqslant 76 \end{aligned} \tag{103}$$

It is only necessary to prove that the increase in the value $c \cdot \frac{2^{q+35-k}}{5^{k+1}} \cdot r$ on the right side of the expression compared to the value $c \cdot \frac{2^{q+35-k}}{5^{k+1}}$ on the left side plus the decimal part of the value on the left side is less than 1 for Equation (99) to hold true. That is:

$$\begin{aligned} \text{float}: \frac{2^{q+34-k} \% 5^{k+1}}{5^{k+1}} + \left(c \cdot \frac{2^{q+35-k}}{5^{k+1}} \cdot r - c \cdot \frac{2^{q+35-k}}{5^{k+1}} \right) < 1 \\ \text{double}: \frac{2^{q+62-k} \% 5^{k+1}}{5^{k+1}} + \left(c \cdot \frac{2^{q+63-k}}{5^{k+1}} \cdot r - c \cdot \frac{2^{q+63-k}}{5^{k+1}} \right) < 1 \end{aligned} \tag{104}$$

 doi:10.20944/preprints202511.1698.v2

By exhaustionally calculating the maximum possible c value under each q and substituting it into Equation (104), it holds. The calculation result is in test2.py. The calculation results show that for the float range and the double range, Equation (104) always holds true. Therefore, Equation (99) holds true, and thus the values of $\lfloor 2^{36} \cdot n \rfloor$ and $\lfloor 2^{64} \cdot n \rfloor$ on the right side of Equations (64) and (65) can be accurately calculated. The values of $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor$ and $\lfloor 2^{63} \cdot 2^q \cdot 10^{-k-1} \rfloor$ on the left side of Equations (64) and (65) can be calculated through lookup tables.

$$\begin{aligned} \text{float}: &\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = pow10 \gg (28 - q - \lfloor (-k-1) \cdot \log_2(10) \rfloor) \\ \text{double}: &\lfloor 2^{63} \cdot 2^q \cdot 10^{-k-1} \rfloor = pow10 \gg (64 - q - \lfloor (-k-1) \cdot \log_2(10) \rfloor) \end{aligned} \tag{105}$$

The code file for verifying the validity of Equation (105) is test4.py. Therefore, when conditions (62) and (63) are met, the values of both sides of Equations (64) and (65) can be accurately calculated.

Discuss the relationship between the following two values within all ranges of floating-point numbers:

$$\begin{aligned} \text{float}: &\lfloor c \cdot 2^{q+36} \cdot 10^{-k-1} \rfloor; \lfloor c \cdot 2^{q+36} \cdot r \cdot 10^{-k-1} \rfloor; \\ \text{double}: &\lfloor c \cdot 2^{q+64} \cdot 10^{-k-1} \rfloor; \lfloor c \cdot 2^{q+64} \cdot r \cdot 10^{-k-1} \rfloor; \end{aligned} \tag{106}$$

When $r = 1$, it is obvious that the two values in expression (106) are equal. When $r \neq 1$, or equivalent to $r > 1$, has:

$$\begin{aligned} \text{float}: c \cdot 2^{q+36} \cdot r \cdot 10^{-k-1} &= c \cdot 2^{q+36} \cdot 10^{-k-1} + c \cdot 2^{q+36} \cdot (r-1) \cdot 10^{-k-1} \\ &< c \cdot 2^{q+36} \cdot 10^{-k-1} + 2^{24} \cdot 2^{36} \cdot 2^q \cdot 10^{-k-1} \cdot (r-1) \\ &< c \cdot 2^{q+36} \cdot 10^{-k-1} + 2^{-3} \\ \lfloor c \cdot 2^{q+36} \cdot r \cdot 10^{-k-1} \rfloor &\leqslant \lfloor c \cdot 2^{q+36} \cdot 10^{-k-1} \rfloor + 1 \\ \text{double}: c \cdot 2^{q+64} \cdot r \cdot 10^{-k-1} &= c \cdot 2^{q+64} \cdot 10^{-k-1} + c \cdot 2^{q+64} \cdot (r-1) \cdot 10^{-k-1} \\ &< c \cdot 2^{q+64} \cdot 10^{-k-1} + 2^{53} \cdot 2^{64} \cdot 2^q \cdot 10^{-k-1} \cdot (r-1) \\ &< c \cdot 2^{q+64} \cdot 10^{-k-1} + 2^{-10} \\ \lfloor c \cdot 2^{q+64} \cdot r \cdot 10^{-k-1} \rfloor &\leqslant \lfloor c \cdot 2^{q+64} \cdot 10^{-k-1} \rfloor + 1 \end{aligned} \tag{107}$$

Therefore, there is:

$$\begin{aligned} \text{float}: &0 \leqslant \lfloor c \cdot 2^{q+36} \cdot r \cdot 10^{-k-1} \rfloor - \lfloor c \cdot 2^{q+36} \cdot 10^{-k-1} \rfloor \leqslant 1 \\ \text{double}: &0 \leqslant \lfloor c \cdot 2^{q+64} \cdot r \cdot 10^{-k-1} \rfloor - \lfloor c \cdot 2^{q+64} \cdot 10^{-k-1} \rfloor \leqslant 1 \end{aligned} \tag{108}$$

Because there is:

$$\lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor c \cdot 2^q \cdot r \cdot 10^{-k-1} \rfloor = m \tag{109}$$

$$\begin{aligned} \text{float}: &\lfloor c \cdot 2^{q+36} \cdot 10^{-k-1} \rfloor = 2^{36} \cdot m + \lfloor 2^{36} \cdot n \rfloor \\ \text{double}: &\lfloor c \cdot 2^{q+64} \cdot 10^{-k-1} \rfloor = 2^{64} \cdot m + \lfloor 2^{64} \cdot n \rfloor \end{aligned} \tag{110}$$

Suppose:

$$n_r = c \cdot 2^q \cdot r \cdot 10^{-k-1} - m \tag{111}$$

Therefore, the following conclusion can be drawn: when condition (62) is met, from Equation (99), we have:

$$\begin{aligned} \text{float}: &2 \leqslant q \leqslant 33 \;\&\&\; (2c \pm 1) \% 5^{k+1} = 0 \Rightarrow \lfloor 2^{36} \cdot n \rfloor = \lfloor 2^{36} \cdot n_r \rfloor \\ \text{double}: &2 \leqslant q \leqslant 76 \;\&\&\; (2c \pm 1) \% 5^{k+1} = 0 \Rightarrow \lfloor 2^{64} \cdot n \rfloor = \lfloor 2^{64} \cdot n_r \rfloor \end{aligned} \tag{112}$$

Within the range of floating-point numbers, there exists:

$$\begin{aligned} \text{float}: &\lfloor 2^{36} \cdot n \rfloor \leqslant \lfloor 2^{36} \cdot n_r \rfloor \leqslant \lfloor 2^{36} \cdot n \rfloor + 1 \\ \text{double}: &\lfloor 2^{64} \cdot n \rfloor \leqslant \lfloor 2^{64} \cdot n_r \rfloor \leqslant \lfloor 2^{64} \cdot n \rfloor + 1 \end{aligned} \tag{113}$$

 doi:10.20944/preprints202511.1698.v2

To simplify the expression, *even* is used to indicate whether c is an even number:

$$even = (c+1)\%2 \in \{0,1\} \tag{114}$$

When $2^{-1}\cdot 2^{q}\cdot 10^{-k-1} = n$ or $2^{-1}\cdot 2^{q}\cdot 10^{-k-1} = 1-n$, $2^{-1}\cdot 2^{q}\cdot 10^{-k-1} = n$ is the boundary condition for $one = 0$, and $2^{-1}\cdot 2^{q}\cdot 10^{-k-1} = 1-n$ is the boundary condition for $one = 10$. Whether *one* is 0 or 10 is determined based on whether c is an even number.Therefore, the following exists:

$$\begin{aligned} \text{float}: &\begin{cases} one = 0 : \lfloor 2^{q+35}\cdot 10^{-k-1}\rfloor + even > \lfloor 2^{36}\cdot n_r\rfloor \\ one = 10 : \lfloor 2^{q+35}\cdot 10^{-k-1}\rfloor + even > 2^{36} - 1 - \lfloor 2^{36}\cdot n_r\rfloor \end{cases} \\ \text{double}: &\begin{cases} one = 0 : \lfloor 2^{q+63}\cdot 10^{-k-1}\rfloor + even > \lfloor 2^{64}\cdot n_r\rfloor \\ one = 10 : \lfloor 2^{q+63}\cdot 10^{-k-1}\rfloor + even > 2^{64} - 1 - \lfloor 2^{64}\cdot n_r\rfloor \end{cases} \end{aligned} \tag{115}$$

Therefore, when $2^{-1}\cdot 2^{q}\cdot 10^{-k-1} = n$ or $2^{-1}\cdot 2^{q}\cdot 10^{-k-1} = 1-n$, we can use the condition (116) to determine whether $one = 0$ or $one = 10$.

$$\begin{aligned} \text{float}: &\begin{cases} \text{if } \lfloor 2^{q+35}\cdot 10^{-k-1}\rfloor + even > \lfloor 2^{36}\cdot n_r\rfloor : one = 0 \\ \text{if } \lfloor 2^{q+35}\cdot 10^{-k-1}\rfloor + even > 2^{36} - 1 - \lfloor 2^{36}\cdot n_r\rfloor : one = 10 \end{cases} \\ \text{double}: &\begin{cases} \text{if } \lfloor 2^{q+63}\cdot 10^{-k-1}\rfloor + even > \lfloor 2^{64}\cdot n_r\rfloor : one = 0 \\ \text{if } \lfloor 2^{q+63}\cdot 10^{-k-1}\rfloor + even > 2^{64} - 1 - \lfloor 2^{64}\cdot n_r\rfloor : one = 10 \end{cases} \end{aligned} \tag{116}$$

When $2^{-1}\cdot 2^{q}\cdot 10^{-k-1} > n$ or $2^{-1}\cdot 2^{q}\cdot 10^{-k-1} > 1-n$, We can also use the above condition (116) to determine whether $one = 0$ or $one = 10$. When $2^{-1}\cdot 2^{q}\cdot 10^{-k-1} < n$ or $2^{-1}\cdot 2^{q}\cdot 10^{-k-1} < 1-n$, we can also use the above condition (116) to determine whether $one \neq 0$ or $one \neq 10$. There are a total of four situations.The proof is as follows:
(1)When $2^{-1}\cdot 2^{q}\cdot 10^{-k-1} < n$, there must exist $one \neq 0$, and there is:

$$\begin{aligned} \text{float}: & 2^{-1}\cdot 2^{q}\cdot 10^{-k-1} - n = n^{-} - 1 \in \left(2^{-33} - 1, -2^{-29}\right) \\ \text{double}: & 2^{-1}\cdot 2^{q}\cdot 10^{-k-1} - n = n^{-} - 1 \in \left(2^{-62} - 1, -2^{-63}\right) \end{aligned} \tag{117}$$

Therefore, the following exists:

$$\begin{aligned} \text{float}: & 2^{q+35}\cdot 10^{-k-1} - 2^{36}\cdot n \in \left(2^{3} - 2^{36}, -2^{7}\right) \\ \text{double}: & 2^{q+63}\cdot 10^{-k-1} - 2^{64}\cdot n \in \left(4 - 2^{64}, -2\right) \end{aligned} \tag{118}$$

Suppose there are two real numbers a and b, and the following relationship must exist:

$$\begin{aligned} 0 \leqslant & b - \lfloor b\rfloor < 1 \\ a - \lfloor a\rfloor - 1 < & b - \lfloor b\rfloor < 1 + a - \lfloor a\rfloor \\ a - b - 1 < & \lfloor a\rfloor - \lfloor b\rfloor < a - b + 1 \end{aligned} \tag{119}$$

When $a = 2^{q+35}\cdot 10^{-k-1}$ and $b = 2^{36}\cdot n$ or $a = 2^{q+63}\cdot 10^{-k-1}$ and $b = 2^{64}\cdot n$, the following exists:

$$\begin{aligned} \text{float}: & \lfloor 2^{q+35}\cdot 10^{-k-1}\rfloor - \lfloor 2^{36}\cdot n\rfloor < 2^{q+35}\cdot 10^{-k-1} - 2^{36}\cdot n + 1 \\ \text{double}: & \lfloor 2^{q+63}\cdot 10^{-k-1}\rfloor - \lfloor 2^{64}\cdot n\rfloor < 2^{q+63}\cdot 10^{-k-1} - 2^{64}\cdot n + 1 \end{aligned} \tag{120}$$

From Equation (118), we have:

$$\begin{aligned} \text{float}: & \lfloor 2^{q+35}\cdot 10^{-k-1}\rfloor - \lfloor 2^{36}\cdot n\rfloor < 1 - 2^{7} < 0 \\ \text{double}: & \lfloor 2^{q+63}\cdot 10^{-k-1}\rfloor - \lfloor 2^{64}\cdot n\rfloor < 1 - 2 < 0 \end{aligned} \tag{121}$$

 doi:10.20944/preprints202511.1698.v2

Therefore, there is:

$$\begin{aligned} \text{float}: \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &\leqslant \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + 1 \\ &< \lfloor 2^{36} \cdot n \rfloor \leqslant \lfloor 2^{36} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &< \lfloor 2^{36} \cdot n_r \rfloor \\ \text{double}: \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &\leqslant \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + 1 \\ &< \lfloor 2^{64} \cdot n \rfloor \leqslant \lfloor 2^{64} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &< \lfloor 2^{64} \cdot n_r \rfloor \end{aligned} \tag{122}$$

Therefore, when $2^{-1} \cdot 2^q \cdot 10^{-k-1} < n$, the condition (116) can be used to determine that $one \neq 0$.
(2)When $2^{-1} \cdot 2^q \cdot 10^{-k-1} > n$, there must exist $one = 0$, and there is:

$$\begin{aligned} \text{float}: & 2^{-1} \cdot 2^q \cdot 10^{-k-1} - n = n^- \in \left(2^{-33}, 1 - 2^{-29}\right) \\ \text{double}: & 2^{-1} \cdot 2^q \cdot 10^{-k-1} - n = n^- \in \left(2^{-62}, 1 - 2^{-63}\right) \end{aligned} \tag{123}$$

Therefore, the following exists:

$$\begin{aligned} \text{float}: & 2^{q+35} \cdot 10^{-k-1} - 2^{36} \cdot n \in \left(2^3, 2^{36} - 2^7\right) \\ \text{double}: & 2^{q+63} \cdot 10^{-k-1} - 2^{64} \cdot n \in \left(4, 2^{64} - 2\right) \end{aligned} \tag{124}$$

When $a = 2^{q+35} \cdot 10^{-k-1}$ and $b = 2^{36} \cdot n$ or $a = 2^{q+63} \cdot 10^{-k-1}$ and $b = 2^{64} \cdot n$, from Equation (119), the following exists:

$$\begin{aligned} \text{float}: & \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor - \lfloor 2^{36} \cdot n \rfloor > 2^{q+35} \cdot 10^{-k-1} - 2^{36} \cdot n - 1 \\ \text{double}: & \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor - \lfloor 2^{64} \cdot n \rfloor > 2^{q+63} \cdot 10^{-k-1} - 2^{64} \cdot n - 1 \end{aligned} \tag{125}$$

From Equation (124), we have:

$$\begin{aligned} \text{float}: & \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor - \lfloor 2^{36} \cdot n \rfloor > 2^3 - 1 \geqslant 0 \\ \text{double}: & \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor - \lfloor 2^{64} \cdot n \rfloor > 4 - 1 \geqslant 0 \end{aligned} \tag{126}$$

Therefore, there is:

$$\begin{aligned} \text{float}: \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &\geqslant \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor \\ &> \lfloor 2^{36} \cdot n \rfloor + 1 \geqslant \lfloor 2^{36} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &> \lfloor 2^{36} \cdot n_r \rfloor \\ \text{double}: \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &\geqslant \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor \\ &> \lfloor 2^{64} \cdot n \rfloor + 1 \geqslant \lfloor 2^{64} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &> \lfloor 2^{64} \cdot n_r \rfloor \end{aligned} \tag{127}$$

Therefore, when $2^{-1} \cdot 2^q \cdot 10^{-k-1} > n$, the condition (116) can be used to determine that $one = 0$.
(3)When $2^{-1} \cdot 2^q \cdot 10^{-k-1} < 1 - n$, there must exist $one \neq 10$, and there is:

$$\begin{aligned} \text{float}: & 2^{-1} \cdot 2^q \cdot 10^{-k-1} + n = n^+ \in \left(2^{-33}, 1 - 2^{-29}\right) \\ \text{double}: & 2^{-1} \cdot 2^q \cdot 10^{-k-1} + n = n^+ \in \left(2^{-62}, 1 - 2^{-63}\right) \end{aligned} \tag{128}$$

 doi:10.20944/preprints202511.1698.v2

Therefore, the following exists:

$$\begin{aligned} \text{float}: & 2^{q+35} \cdot 10^{-k-1} + 2^{36} \cdot n \in \left(2^3, 2^{36} - 2^7\right) \\ \text{double}: & 2^{q+63} \cdot 10^{-k-1} + 2^{64} \cdot n \in \left(4, 2^{64} - 2\right) \end{aligned} \tag{129}$$

Suppose there are two real numbers a and b, and the following relationship must exist:

$$\begin{aligned} a - 1 < \lfloor a \rfloor \leqslant a \\ b - 1 < \lfloor b \rfloor \leqslant b \\ a + b - 2 < \lfloor a \rfloor + \lfloor b \rfloor \leqslant a + b \end{aligned} \tag{130}$$

When $a = 2^{q+35} \cdot 10^{-k-1}$ and $b = 2^{36} \cdot n$ or $a = 2^{q+63} \cdot 10^{-k-1}$ and $b = 2^{64} \cdot n$, the following exists:

$$\begin{aligned} \text{float}: & \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + \lfloor 2^{36} \cdot n \rfloor \leqslant 2^{q+35} \cdot 10^{-k-1} + 2^{36} \cdot n \\ \text{double}: & \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + \lfloor 2^{64} \cdot n \rfloor \leqslant 2^{q+63} \cdot 10^{-k-1} + 2^{64} \cdot n \end{aligned} \tag{131}$$

From Equation (129), we have:

$$\begin{aligned} \text{float}: & \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + \lfloor 2^{36} \cdot n \rfloor < 2^{36} - 2^7 \\ \text{double}: & \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + \lfloor 2^{64} \cdot n \rfloor < 2^{64} - 2 \end{aligned} \tag{132}$$

Therefore, there is:

$$\begin{aligned} \text{float}: \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &\leqslant \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + 1 \\ &< 2^{36} - 2 - \lfloor 2^{36} \cdot n \rfloor \\ &< 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &< 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor \\ \text{double}: \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &\leqslant \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + 1 \\ &< 2^{64} - 2 - \lfloor 2^{64} \cdot n \rfloor \\ &< 2^{64} - 1 - \lfloor 2^{64} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &< 2^{64} - 1 - \lfloor 2^{64} \cdot n_r \rfloor \end{aligned} \tag{133}$$

Therefore, when $2^{-1} \cdot 2^q \cdot 10^{-k-1} < 1 - n$, the condition (116) can be used to determine that $one \neq 10$.
(4)When $2^{-1} \cdot 2^q \cdot 10^{-k-1} > 1 - n$, there must exist $one = 10$, and there is:

$$\begin{aligned} \text{float}: & 2^{-1} \cdot 2^q \cdot 10^{-k-1} + n = n^+ + 1 \in \left(1 + 2^{-33}, 2 - 2^{-29}\right) \\ \text{double}: & 2^{-1} \cdot 2^q \cdot 10^{-k-1} + n = n^+ + 1 \in \left(1 + 2^{-62}, 2 - 2^{-63}\right) \end{aligned} \tag{134}$$

Therefore, the following exists:

$$\begin{aligned} \text{float}: & 2^{q+35} \cdot 10^{-k-1} + 2^{36} \cdot n \in \left(2^3 + 2^{36}, 2^{37} - 2^7\right) \\ \text{double}: & 2^{q+63} \cdot 10^{-k-1} + 2^{64} \cdot n \in \left(4 + 2^{64}, 2^{65} - 2\right) \end{aligned} \tag{135}$$

When $a = 2^{q+35} \cdot 10^{-k-1}$ and $b = 2^{36} \cdot n$ or $a = 2^{q+63} \cdot 10^{-k-1}$ and $b = 2^{64} \cdot n$, from Equation (130), the following exists:

$$\begin{aligned} \text{float}: & \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + \lfloor 2^{36} \cdot n \rfloor > 2^{q+35} \cdot 10^{-k-1} + 2^{36} \cdot n - 2 \\ \text{double}: & \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + \lfloor 2^{64} \cdot n \rfloor > 2^{q+63} \cdot 10^{-k-1} + 2^{64} \cdot n - 2 \end{aligned} \tag{136}$$

 doi:10.20944/preprints202511.1698.v2

From Equation (135), we have:

$$\begin{aligned} \text{float}: &\lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + \lfloor 2^{36} \cdot n \rfloor > 2^{36} + 2^3 - 2 > 2^{36} \\ \text{double}: &\lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + \lfloor 2^{64} \cdot n \rfloor > 2^{64} + 2 - 2 > 2^{64} \end{aligned} \tag{137}$$

Therefore, there is:

$$\begin{aligned} \text{float}: \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &\geqslant \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor \\ &> 2^{36} - \lfloor 2^{36} \cdot n \rfloor \\ &> 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &> 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor \\ \text{double}: \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &\geqslant \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor \\ &> 2^{64} - \lfloor 2^{64} \cdot n \rfloor \\ &> 2^{64} - 1 - \lfloor 2^{64} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &> 2^{64} - 1 - \lfloor 2^{64} \cdot n_r \rfloor \end{aligned} \tag{138}$$

Therefore, when $2^{-1} \cdot 2^q \cdot 10^{-k-1} > 1 - n$, the condition (116) can be used to determine that $one = 10$.

From the above proof, it can be seen that when condition (62) is met, the condition (116) can be used to determine whether $one = 0$ or $one = 10$ when $2^{-1} \cdot 2^q \cdot 10^{-k-1} = n$ or $2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n$. When $2^{-1} \cdot 2^q \cdot 10^{-k-1} > n$ or $2^{-1} \cdot 2^q \cdot 10^{-k-1} > 1 - n$, the condition (116) can be used to determine whether $one = 0$ or $one = 10$. When $2^{-1} \cdot 2^q \cdot 10^{-k-1} < n$ or $2^{-1} \cdot 2^q \cdot 10^{-k-1} < 1 - n$, the condition (116) can be used to determine whether $one \neq 0$ or $one \neq 10$.

The proof process of this section is completed. In the code implementation, the two judgment conditions can be quickly calculated using addition and subtraction shift operations, and can be compiled by the compiler into cmov instructions, thereby reducing the impact of branch prediction failure on performance.

Ultimately, we reached the following conclusion: *one* can quickly determine whether *one* equals 0 or 10 by using the following method.

$$\begin{aligned} \text{float}: &\lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even > \lfloor 2^{36} \cdot n_r \rfloor \Rightarrow one = 0 \\ &\lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even > 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor \Rightarrow one = 10 \\ \text{double}: &\lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even > \lfloor 2^{64} \cdot n_r \rfloor \Rightarrow one = 0 \\ &\lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even > 2^{64} - 1 - \lfloor 2^{64} \cdot n_r \rfloor \Rightarrow one = 10 \end{aligned} \tag{139}$$

3.7. Efficient Computation of $\lfloor 10n \rfloor$ and Rounding

Determine whether *one* is $\lfloor 10n \rfloor$ or $\lfloor 10n \rfloor + 1$ based on the decimal part of $10n$. There are two cases: the decimal part of $10n$ is 0.5 and it is not 0.5.

3.7.1. $10n - \lfloor 10n \rfloor = 0.5$

When the decimal part of $10n$ is 0.5, there must be:

$$\begin{aligned} 10n - \lfloor 10n \rfloor &= 0.5 \\ \Rightarrow 10 \cdot c \cdot 2^q \cdot 10^{-k-1} - \lfloor 10 \cdot c \cdot 2^q \cdot 10^{-k-1} \rfloor &= 0.5 \\ \Rightarrow c \cdot 2^q \cdot 10^{-k} - \lfloor c \cdot 2^q \cdot 10^{-k} \rfloor &= 0.5 \\ \Rightarrow c \cdot 2^q \cdot 10^{-k} &= \lfloor c \cdot 2^q \cdot 10^{-k} \rfloor + 0.5 \\ \Rightarrow 2c \cdot 2^q \cdot 10^{-k} &= 2\lfloor c \cdot 2^q \cdot 10^{-k} \rfloor + 1 \end{aligned} \tag{140}$$

So $2c \cdot 2^q \cdot 10^{-k}$ is an odd number. Then the following expression is odd:

$$c \cdot 2^{q+1} \cdot 10^{-k} = c \cdot 2^{q-k+1} \cdot 5^{-k} \tag{141}$$

 doi:10.20944/preprints202511.1698.v2

According to the range of q, there are:

$$c \cdot 2^{q+1} \cdot 10^{-k} = \begin{cases} \frac{c \cdot 2^{q-k+1}}{5^k}; q \geqslant 0 \\ c \cdot 2 \cdot 5^{-k}; q = -1 \\ \frac{c \cdot 5^{-k}}{2^{k-q-1}}; q \leqslant -2 \end{cases} \tag{142}$$

According to the range of q, the following situations are discussed:

- $q \geqslant 0$
  When $q \geqslant 0$, it can be concluded that $q - k + 1 \geqslant 1$, the numerator $c \cdot 2^{q-k+1}$ is even and the denominator 5^k is odd, which does not meet the condition.
- $q = -1$
  When $q = -1$, it can be concluded that $c \cdot 2 \cdot 5^{-k}$ is even, which does not meet the condition.
- $q \leqslant -2$
  5^{-k} is an odd number. c is an odd multiple of 2^{k-q-1}. So:

$$\begin{aligned} \text{float} &: c \geqslant 2^{k-q-1} \Rightarrow k - q - 1 \leqslant 22 \Rightarrow q \geqslant -34 \\ \text{double} &: c \geqslant 2^{k-q-1} \Rightarrow k - q - 1 \leqslant 51 \Rightarrow q \geqslant -75 \end{aligned} \tag{143}$$

  Therefore, when q meets the above conditions, c must be an odd multiple of 2^{k-q-1} to meet the condition. Therefore, when the following conditions are met, expression (141) is an odd number:

$$\begin{aligned} \text{float} &: -34 \leqslant q \leqslant -2 \,\&\&\, c\%2^{k-q} = 2^{k-q-1} \\ \text{double} &: -75 \leqslant q \leqslant -2 \,\&\&\, c\%2^{k-q} = 2^{k-q-1} \end{aligned} \tag{144}$$

  When q is within the above range (144), $r = 1$ is derived from Equation (28).Therefore, there is:

$$n_r = n \tag{145}$$

  The following equation holds:

$$20m + 20n = c \cdot 2^q \cdot 10^{-k+1} = c \cdot 2^{q-k+1} \cdot 5^{-k} = \frac{c}{2^{k-q-1}} \cdot 5^{-k} \tag{146}$$

  Since $-k \geqslant 1$, 5^{-k} is multiple of 5 and is an odd number. Since $\frac{c}{2^{k-q-1}}$ and 5^{-k} are both odd numbers, $20m$ is an even number, $20n$ is multiple of 5 and is an odd number. Therefore, there is:

$$\begin{aligned} & 20n \in \{5, 15\} \\ \Rightarrow\ & n \in \{0.25, 0.75\} \\ \Rightarrow\ & n_r \in \{0.25, 0.75\} \end{aligned} \tag{147}$$

  The result of *one* is an even number between $\lfloor 10n \rfloor$ and $\lfloor 10n \rfloor + 1$. Therefore:

$$one = \begin{cases} \lfloor 10n \rfloor = 2, \text{if } n = 0.25 \\ \lfloor 10n \rfloor + 1 = 8, \text{if } n = 0.75 \end{cases} \Rightarrow one = \lfloor 20n + 1 \rfloor // 2 - (n = 0.25\,?1 : 0) \tag{148}$$

3.7.2. $10n - \lfloor 10n \rfloor \neq 0.5$

When the decimal part of $10n$ is not 0.5, round to the nearest integer value based on the decimal part of $10n$. Therefore, there is:

$$one = \begin{cases} \lfloor 10n \rfloor, \text{if } 10n - \lfloor 10n \rfloor < 0.5 \\ \lfloor 10n \rfloor + 1, \text{if } 10n - \lfloor 10n \rfloor > 0.5 \end{cases} \Rightarrow one = \lfloor 10n + 0.5 \rfloor = \lfloor 20n + 1 \rfloor // 2 \tag{149}$$

 doi:10.20944/preprints202511.1698.v2

Since $\lfloor 20n+1 \rfloor = \lfloor 20n \rfloor + 1$, it is only necessary to accurately calculate the value of $\lfloor 20n \rfloor$. And, there is:

$$\begin{aligned} d &= ten + one \\ &= 10m + \lfloor 20n+1 \rfloor //2 \\ &= (\lfloor 20m + 20n \rfloor + 1)//2 \end{aligned} \tag{150}$$

Suppose there are:

$$20m + 20n = c \cdot 2^{q+1} \cdot 10^{-k} = c \cdot 2^{q-k+1} \cdot 5^{-k} = c \cdot \frac{x}{y} \tag{151}$$

Suppose the decimal part of $20n$ is n_{20}.

When $y \leqslant c_{\max} = C$, the range of the decimal part must include:

$$\begin{aligned} \text{float}: & \frac{1}{2^{24}-1} = \frac{1}{C} \leqslant n_{20} \leqslant 1 - \frac{1}{C} = \frac{2^{24}-2}{2^{24}-1} \\ \text{double}: & \frac{1}{2^{53}-1} = \frac{1}{C} \leqslant n_{20} \leqslant 1 - \frac{1}{C} = \frac{2^{53}-2}{2^{53}-1} \end{aligned} \tag{152}$$

When $y > c_{\max} = C$, the range of the decimal part must include(the test file is test5.py):

$$\begin{aligned} \text{float}: & 2^{-32} < n_{20} < 1 - 2^{-30} \\ \text{double}: & 2^{-64} < n_{20} < 1 - 2^{-62} \end{aligned} \tag{153}$$

Therefore, the range of n_{20} satisfies Equation (153). In the code implementation, for float, only the high 36 bits of n_r are retained, and for double, only the high 70 bits of n_r are retained. Suppose the discarded part of a float is represented as n_{36}, and similarly, the discarded part of a double is represented as n_{70}. Therefore, there is:

$$\begin{aligned} \text{float}: & n_{36} \in \left[0, 2^{-36}\right) \\ \text{double}: & n_{70} \in \left[0, 2^{-70}\right) \end{aligned} \tag{154}$$

Calculate the boundary conditions of the following expression:

$$\begin{aligned} \text{float}: & F = 20 \cdot \left(c \cdot 2^q \cdot r \cdot 10^{-k-1} - n_{36}\right) \\ \text{double}: & F = 20 \cdot \left(c \cdot 2^q \cdot r \cdot 10^{-k-1} - n_{70}\right) \end{aligned} \tag{155}$$

Therefore, there is:

$$\begin{aligned} \text{float}: F_{\min} &> 20 \cdot \left(c \cdot 2^q \cdot 10^{-k-1} - 2^{-36}\right) \\ &= 20m + 20n - 20 \cdot 2^{-36} \\ F_{\max} &< 20 \cdot \left(c \cdot 2^q \cdot \left(1 + 2^{-63}\right) \cdot 10^{-k-1} - 0\right) \\ &< 20m + 20n + 20 \cdot 2^{-63} \cdot c \\ &< 20m + \lfloor 20n \rfloor + 1 \\ \text{double}: F_{\min} &> 20 \cdot \left(c \cdot 2^q \cdot 10^{-k-1} - 2^{-70}\right) \\ &= 20m + 20n - 20 \cdot 2^{-70} \\ &> 20m + \lfloor 20n \rfloor \\ F_{\max} &< 20 \cdot \left(c \cdot 2^q \cdot \left(1 + 2^{-127}\right) \cdot 10^{-k-1} - 0\right) \\ &< 20m + 20n + 20 \cdot 2^{-127} \cdot c \\ &< 20m + \lfloor 20n \rfloor + 1 \end{aligned} \tag{156}$$

 doi:10.20944/preprints202511.1698.v2

Therefore, there is:

$$\begin{aligned} \text{float}: \lfloor F \rfloor &= 20m + \lfloor 20n \rfloor \\ \text{double}: \lfloor F \rfloor &= 20m + \lfloor 20n \rfloor \end{aligned} \tag{157}$$

In fact, in the above proof process, for float, $\lfloor F_{min} \rfloor \neq 20m + \lfloor 20n \rfloor$ may exist, but the code implementation has passed the exhaustive test, so this not-so-perfect proof process can be ignored. Therefore, the calculation of d can be simplified as follows:

$$\begin{aligned} d &= ten + one \\ &= (\lfloor F \rfloor + 1)//2 \\ &= (\lfloor 20 \cdot (c \cdot 2^q \cdot r \cdot 10^{-k-1} - n_x) \rfloor + 1)//2 \end{aligned} \tag{158}$$

For the float range, $n_x = n_{36}$; for the double range, $n_x = n_{70}$.

3.7.3. Efficient Implementation of $n = 0.25$ for Double

For double, quickly determine that $n == 0.25$ in Equation (148).
When $n = 0.25, \lfloor 2^{64} \cdot n_r \rfloor = \lfloor 2^{64} \cdot n \rfloor = 2^{62}$. Therefore, the following condition can be used to quickly determine whether $n = 0.25$:

$$\text{double}: n = 0.25 \text{ if } \lfloor 2^{64} \cdot n_r \rfloor = 2^{62} \tag{159}$$

When $n \neq 0.25$,Calculate the range of the decimal part of the following expression:

$$4m + 4n = c \cdot 2^{q+2} \cdot 10^{-k-1} \tag{160}$$

Therefore, when Equation (160) is not an integer, we have(the test file is test6.py):

$$2^{-62} < 4n - \lfloor 4n \rfloor < 1 - 2^{-62} \tag{161}$$

Calculate the two boundary cases of $4n$ that are closest to 1:

$$\begin{aligned} \lfloor 4n \rfloor = 0 \Rightarrow 4n - 0 < 1 - 2^{-62} \Rightarrow \lfloor 2^{64} \cdot n \rfloor \leqslant 2^{62} - 2 \\ \lfloor 4n \rfloor = 1 \Rightarrow 4n - 1 > 2^{-62} \Rightarrow \lfloor 2^{64} \cdot n \rfloor \geqslant 2^{62} + 1 \end{aligned} \tag{162}$$

Then there are:

$$\begin{aligned} \lfloor 2^{64} \cdot n \rfloor \neq 2^{62} \,\&\&\, \lfloor 2^{64} \cdot n \rfloor + 1 \neq 2^{62} \\ \Rightarrow \lfloor 2^{64} \cdot n_r \rfloor \neq 2^{62} \end{aligned} \tag{163}$$

Therefore, the following condition can be used to quickly determine whether $n \neq 0.25$:

$$\text{double}: n \neq 0.25 \text{ if } \lfloor 2^{64} \cdot n_r \rfloor \neq 2^{62} \tag{164}$$

In summary, for double, the following condition can be used to quickly determine whether $n = 0.25$:

$$\begin{aligned} \text{double}: n = 0.25 \text{ if } \lfloor 2^{64} \cdot n_r \rfloor = 2^{62} \\ \text{double}: n \neq 0.25 \text{ if } \lfloor 2^{64} \cdot n_r \rfloor \neq 2^{62} \end{aligned} \tag{165}$$

3.7.4. Efficient Calculation of *one* for Double

In the double range, introduce another faster way to calculate *one*:

$$\text{double}: one = \lfloor \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 + \left((n = 0.25)?0 : \left(2^{-1} + \frac{6}{2^{64}} \right) \right) \rfloor \tag{166}$$

 doi:10.20944/preprints202511.1698.v2

The proof of Equation (166) is as follows:
when $n = 0.25$, $\lfloor \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 \rfloor = \lfloor 10n \rfloor = 2$;
when $n \neq 0.25$, Equation (166) can be equivalent to the following:

$$\text{double} : one = \lfloor \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 + 2^{-1} + \frac{6}{2^{64}} \rfloor \tag{167}$$

According to the $10n - \lfloor 10n \rfloor$ range, one is represented as:

$$\text{double} : one = \begin{cases} \lfloor 10n \rfloor, \text{if } 10n - \lfloor 10n \rfloor < 0.5 \\ 8, \text{if } 10n - \lfloor 10n \rfloor = 0.5 \\ \lfloor 10n \rfloor + 1, \text{if } 10n - \lfloor 10n \rfloor > 0.5 \end{cases} = \lfloor 20n + 1 \rfloor // 2 \tag{168}$$

Therefore,when $n \neq 0.25$, we need to prove that the following equation holds:

$$\lfloor \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 + 2^{-1} + \frac{6}{2^{64}} \rfloor = \begin{cases} \lfloor 10n \rfloor, \text{if } 10n - \lfloor 10n \rfloor < 0.5 \\ 8, \text{if } 10n - \lfloor 10n \rfloor = 0.5 \\ \lfloor 10n \rfloor + 1, \text{if } 10n - \lfloor 10n \rfloor > 0.5 \end{cases} = \lfloor 20n + 1 \rfloor // 2 \tag{169}$$

From the range of n, there is:

$$\frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \in \left(n_r - 2^{-64}, n_r \right] \tag{170}$$

Because the following conditions exist:

$$\begin{aligned} c \cdot 2^q \cdot 10^{-k-1} &= m + n \\ c \cdot 2^q \cdot r \cdot 10^{-k-1} &= m + n_r \end{aligned} \tag{171}$$

Therefore, the following relationship can be concluded:

$$\begin{aligned} n_r - n &= (r-1) \cdot c \cdot 2^q \cdot 10^{-k-1} \\ n_r &= (r-1) \cdot (m+n) + n \\ \Rightarrow n \leqslant n_r &< 2^{-127} \cdot c + n \\ n \leqslant n_r &< 2^{-127} \cdot 2^{53} + n \\ n \leqslant n_r &< 2^{-74} + n \end{aligned} \tag{172}$$

From Equation (170) and (172), it can be concluded that:

$$\begin{aligned} \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} &\in \left(n - 2^{-64}, n + 2^{-74} \right) \\ \Rightarrow \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 &\in \left(10n - 10 \cdot 2^{-64}, 10n + 10 \cdot 2^{-74} \right) \\ \Rightarrow \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 20 &\in \left(20n - 20 \cdot 2^{-64}, 20n + 20 \cdot 2^{-74} \right) \\ \Rightarrow \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 20 &\in \left(\lfloor 20n \rfloor + n_{20} - 20 \cdot 2^{-64}, \lfloor 20n \rfloor + n_{20} + 20 \cdot 2^{-74} \right) \end{aligned} \tag{173}$$

Discuss the range of values of x when the following conditions are met.

$$\lfloor \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 20 + 1 + x \rfloor // 2 = \lfloor 20n + 1 \rfloor // 2 = one \tag{174}$$

 doi:10.20944/preprints202511.1698.v2

Therefore, the following conclusions can be drawn:

$$\begin{aligned} \lfloor 20n \rfloor + n_{20} - 20 \cdot 2^{-64} + 1 + x \geqslant \lfloor 20n + 1 \rfloor &\Rightarrow x \geqslant 20 \cdot 2^{-64} - n_{20} \\ \lfloor 20n \rfloor + n_{20} + 20 \cdot 2^{-74} + 1 + x < \lfloor 20n + 2 \rfloor &\Rightarrow x < 1 - 20 \cdot 2^{-74} - n_{20} \end{aligned} \tag{175}$$

Suppose $x = 12 \cdot 2^{-64}$. Through the exhaustive method, all floating-point numbers that do not meet the following conditions can be obtained.

$$x = 12 \cdot 2^{-64} \geqslant 20 \cdot 2^{-64} - n_{20} \tag{176}$$

All floating-point numbers that do not meet condition (176) are as follows (in hexadecimal) :

$$\begin{gathered} 0xd17c0747bd76fa1, \\ 0xd27c0747bd76fa1, \\ 0x4d73de005bd620df, \\ 0x4d83de005bd620df, \\ 0x4d93de005bd620df, \end{gathered} \tag{177}$$

Through the exhaustive method, all floating-point numbers that do not meet the following conditions can be obtained.

$$x = 12 \cdot 2^{-64} < 1 - 20 \cdot 2^{-74} - n_{20} \tag{178}$$

All floating-point numbers that do not meet condition (178) are as follows (in hexadecimal) :

$$\begin{gathered} 0x612491daad0ba280, \\ 0x6159b651584e8b20, \\ 0x619011f2d73116f4, \\ 0x61c4166f8cfd5cb1, \\ 0x61d4166f8cfd5cb1, \end{gathered} \tag{179}$$

There are:

$$2(\frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 + 2^{-1} + \frac{6}{2^{64}}) = \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 20 + 1 + x \tag{180}$$

When the floating-point number is not within the above range (177) and (179), the condition (175) is satisfied. We have tested all floating-point numbers within the above-mentioned range (177) and (179), and the algorithm implementation code has output the correct result, that is, it satisfies the SW principle. The test process file is test8.py.

In summary, Equation (169) and Equation (166) holds. Therefore, Equation (166) can be used to quickly calculate *one*.

3.8. Irregular Number

Due to the limited and small number of irregular floating-point numbers, there are a total of 2046 double floating-point numbers and 254 float floating-point numbers. The correctness of the algorithm code in this paper can be proved by the exhaustive method. Therefore, it is not introduced in this article. For the specific implementation process, please refer to the source code.

3.9. Implementation of Pseudocode

This subsection details the pseudocode implementation for converting regular floating-point numbers to decimal representation. The handling of subnormal (irregular) floating-point numbers is omitted here due to space constraints; interested readers may refer to the source code for the complete implementation.

 doi:10.20944/preprints202511.1698.v2

3.9.1. Single-Precision Floating-Point Numbers

$$\begin{aligned}
\text{uint32 } vi &= \text{bit_copy_from_f32}(v) \\
(\text{uint64 } c, \text{int32 } q) &= \text{extract}(vi) \\
\text{const int BIT} &= 36 \\
\text{const uint64 offset} &= (1\text{ULL} \ll (\text{BIT} - 2)) - 7 \\
\text{int32 } k &= (q \cdot 1233) \gg 12 \\
\text{int32 } h &= q + ((1701 \cdot (-k - 1)) \gg 9) \\
\text{uint64 pow10} &= \text{get_pow10}(-k - 1) \\
\text{uint64 } cb &= c \ll (h + \text{BIT} + 1) \\
\text{uint64 hi64} &= \text{umul_hi}_{64\times 64}(cb, \text{pow10}) \\
\text{bool even} &= (vi + 1)\&1 \\
\text{uint64 half} &= (\text{pow10} \gg (65 - (h + \text{BIT} + 1))) + \text{even} \\
\text{uint32 shorter} &= (\text{hi64} + \text{half}) \gg \text{BIT} \\
\text{uint64 bias} &= (\text{hi64} \gg (\text{BIT} - 4))\&15 \\
\text{uint32 longer} &= (5 \cdot \text{hi64} + \text{offset} + \text{bias}) \gg (\text{BIT} - 1) \\
\text{bool updown} &= \text{shorter} > ((\text{hi64} - \text{half}) \gg \text{BIT}) \\
\text{uint32 } d &= \text{updown ? shorter} \cdot 10 : \text{longer}
\end{aligned} \tag{181}$$

Algorithm 181 outlines the procedure for computing d and k for single-precision floating-point numbers.

Since c contains at most 23 significant bits and pow10 has 64 significant bits, their product contains at most 87 significant bits. Given that the range of h is $[-4, -1]$, the value $cb = c \ll (h + \text{BIT} + 1)$ does not overflow, preserving all bits of c. When $e_{10} = -k - 1$, pow10 is computed using Equation (24). From Equation (25), we derive:

$$\text{pow10} \cdot 2^{\lfloor (-k-1)\cdot \log_2 10 \rfloor - 63} = 10^{-k-1} \cdot r_{1,-k-1} \tag{182}$$

From Equation (51):

$$\begin{aligned}
m &= \lfloor v \cdot 10^{-k-1} \rfloor \\
&= \lfloor v \cdot 10^{-k-1} \cdot r_{1,-k-1} \rfloor \\
&= \lfloor v \cdot \text{pow10} \cdot 2^{\lfloor (-k-1)\cdot \log_2 10 \rfloor - 63} \rfloor \\
&= \lfloor c \cdot \text{pow10} \cdot 2^{q + \lfloor (-k-1)\cdot \log_2 10 \rfloor - 63} \rfloor \\
&= (c \cdot \text{pow10}) \gg (63 - q - \lfloor (-k-1) \cdot \log_2 10 \rfloor) \\
&= (c \cdot \text{pow10}) \gg (63 - h) \\
&= ((c \ll (37 + h)) \cdot \text{pow10}) \gg (36 + 64) \\
&= \text{hi64} \gg 36
\end{aligned} \tag{183}$$

Let up indicate whether $one = 10$. From Equations (138) and (105):

$$\begin{aligned}
\text{bool up} &= \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + \text{even} > 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor \\
&= \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + \text{even} + \lfloor 2^{36} \cdot n_r \rfloor \geqslant 2^{36} \\
&= (\text{pow10} \gg (28 - h)) + \text{even} + \lfloor 2^{36} \cdot n_r \rfloor \geqslant 2^{36} \\
&= (\text{half} + \lfloor 2^{36} \cdot n_r \rfloor) \gg 36
\end{aligned} \tag{184}$$

 doi:10.20944/preprints202511.1698.v2

When $one \neq 10$, we have $d//10 = m$; when $one = 10$, we have $d//10 = m + 1$. Therefore:

$$d//10 = m + \text{up} \tag{185}$$

The upper 28 bits of hi64 represent m, while the lower 36 bits represent $\lfloor 2^{36} \cdot n_r \rfloor$:

$$\text{hi64} = (m \ll 36) + \lfloor 2^{36} \cdot n_r \rfloor \tag{186}$$

Consequently:

$$\text{shorter} = d//10 = (\text{hi64} + \text{half}) \gg 36 \tag{187}$$

Let updown indicate whether $one \in \{0, 10\}$, equivalently whether the last digit of d is zero:

$$\text{updown} = (d \bmod 10 = 0) \tag{188}$$

For brevity, let $\text{dot\_one} = \lfloor 2^{36} \cdot n_r \rfloor$. From Equations (134) and (140):

$$\begin{aligned} one = 0 &: \text{half} > \text{dot\_one} \\ one = 10 &: \text{half} > 2^{36} - 1 - \text{dot\_one} \end{aligned} \tag{189}$$

The following equations hold:

$$\begin{aligned} one = 0 &: (\text{hi64} - \text{half}) \gg 36 = m - 1 \\ one = 10 &: (\text{hi64} - \text{half}) \gg 36 = m \end{aligned} \tag{190}$$

Therefore:

$$\begin{aligned} one = 0 &: \text{shorter} = m > (\text{hi64} - \text{half}) \gg 36 = m - 1 \\ one = 10 &: \text{shorter} = m + 1 > (\text{hi64} - \text{half}) \gg 36 = m \end{aligned} \tag{191}$$

Thus, the condition for $one \in \{0, 10\}$ is:

$$\text{shorter} > (\text{hi64} - \text{half}) \gg 36 \implies d \bmod 10 = 0 \tag{192}$$

When updown is true, $d = 10 \cdot \text{shorter}$. Otherwise, we compute d using Equation (158). When $10n - \lfloor 10n \rfloor \neq 0.5$:

$$\begin{aligned} d &= (\lfloor 20 \cdot (v \cdot 10^{-k-1} \cdot r - n_{36}) \rfloor + 1)//2 \\ &= (\lfloor 20 \cdot (\text{hi64} \cdot 2^{-36}) \rfloor + 1)//2 \\ &= (\lfloor 5 \cdot \text{hi64} \cdot 2^{-34} \rfloor + 1)//2 \\ &= (\lfloor (5 \cdot \text{hi64} + 2^{34}) \cdot 2^{-34} \rfloor)//2 \\ &= ((5 \cdot \text{hi64} + 2^{34}) \gg 34)//2 \\ &= (5 \cdot \text{hi64} + 2^{34}) \gg 35 \end{aligned} \tag{193}$$

Adding 2^{34} serves as a rounding operation. When $10n - \lfloor 10n \rfloor = 0.5$, Equation (148) requires checking whether $n = 0.25$. The derivation of longer involves careful handling of edge cases; the correctness of this computation has been verified through exhaustive testing.

 doi:10.20944/preprints202511.1698.v2

3.9.2. Double-Precision Floating-Point Numbers

$$\begin{aligned}
\text{uint64 } vi &= \text{bit_copy_from_f64}(v) \\
(\text{uint64 } c, \text{int } q) &= \text{extract}(vi) \\
\text{const int BIT} &= 6 \\
\text{int } k &= (q \cdot 78913) \gg 18 \\
\text{int } h &= q + ((217707 \cdot (-k-1)) \gg 16) \\
\text{uint128 pow10} &= \text{get_pow10}(-k-1) \\
\text{uint64 } cb &= c \ll (h + \text{BIT} + 1) \\
\text{uint128 hi128} &= \text{umul_hi}_{64\times128}(cb, \text{pow10}) \\
\text{bool even} &= (vi+1)\&1 \\
\text{uint64 half} &= (\text{pow10} \gg (64-h)) + \text{even} \\
\text{uint64 dot_one} &= (\text{uint64})(\text{hi128} \gg \text{BIT}) \\
\text{uint64 ten} &= 10 \cdot (\text{hi128} \gg (\text{BIT}+64)) \\
\text{uint64 offset_num} &= (\text{dot_one} == 2^{62})\,?\,0 : 2^{63}+6 \\
\text{uint64 } one &= ((\text{uint128})10 \cdot \text{dot_one} + \text{offset_num}) \gg 64 \\
one &= (\text{half} > \text{dot_one})\,?\,0 : one \\
one &= (\text{half} > 2^{64} - 1 - \text{dot_one})\,?\,10 : one \\
\text{uint64 } d &= \text{ten} + one
\end{aligned} \tag{194}$$

Algorithm 194 presents the corresponding procedure for double-precision floating-point numbers. pow10 is computed using Equation (26), satisfying:

$$\text{pow10} \cdot 2^{\lfloor(-k-1)\cdot\log_2 10\rfloor - 127} = 10^{-k-1} \cdot r_{1,-k-1} \tag{195}$$

From Equation (51):

$$\begin{aligned}
m &= \lfloor v \cdot 10^{-k-1} \rfloor \\
&= \lfloor v \cdot 10^{-k-1} \cdot r_{1,-k-1} \rfloor \\
&= \lfloor v \cdot \text{pow10} \cdot 2^{\lfloor(-k-1)\cdot\log_2 10\rfloor - 127} \rfloor \\
&= \lfloor c \cdot \text{pow10} \cdot 2^{q + \lfloor(-k-1)\cdot\log_2 10\rfloor - 127} \rfloor \\
&= (c \cdot \text{pow10}) \gg (127 - q - \lfloor(-k-1)\cdot\log_2 10\rfloor) \\
&= (c \cdot \text{pow10}) \gg (127 - h) \\
&= ((c \ll (7+h)) \cdot \text{pow10}) \gg (64+70) \\
&= \text{hi128} \gg 70
\end{aligned} \tag{196}$$

Given $h \in [-4, -1]$ and c having at most 53 significant bits, cb does not overflow, ensuring accurate computation of ten $= 10 \cdot m$. By definition, dot_one $= \lfloor 2^{64} \cdot n_r \rfloor$. From Equation (105):

$$\begin{aligned}
\text{half} &= (\text{pow10} \gg (64-h)) + \text{even} \\
&= \lfloor 2^{63} \cdot 2^q \cdot 10^{-k-1} \rfloor + \text{even}
\end{aligned} \tag{197}$$

The value of one is first computed using Equation (166), then adjusted if the conditions for $one = 0$ or $one = 10$ are met:

$$\begin{aligned}
one &= \left\lfloor \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 + \left((n = 0.25)\,?\,0 : (2^{-1} + \frac{6}{2^{64}}) \right) \right\rfloor \\
&= (\text{dot_one} \cdot 10 + (\text{dot_one} == 2^{62}\,?\,0 : 2^{63}+6)) \gg 64
\end{aligned} \tag{198}$$

 doi:10.20944/preprints202511.1698.v2

An equivalent formulation is:

$$one = (\text{dot\_one} == 2^{62})\,?\,2 : (\text{dot\_one} \cdot 10 + 2^{63} + 6) \gg 64 \tag{199}$$

The conditions $one = 0$ and $one = 10$ are determined by Equations (127) and (138), respectively.

In the C/C++ implementation, the only branch occurs when computing c and q for subnormal floating-point numbers (marked as unlikely). The conversion of normal floating-point numbers is branch-free, eliminating branch misprediction penalties. As shown in Algorithms 181 and 194, the core algorithm is remarkably concise. Irregular numbers are handled in a separate branch—a worthwhile trade-off given their rarity.

3.10. Decimal to String Conversion

The floating-point number printing algorithm consists of two main phases. While the first phase, which computes the decimal digits d and exponent k, has been discussed in previous sections, this section focuses on the second phase: converting the computed decimal representation into a string.

Floating-point numbers are typically printed in two formats: fixed-point notation and scientific notation. Table (3) illustrates examples of both formats.

Table 3. Example of floating-point number printing result.

float number	fixed-point	scientific
2.34	“2.34”	“2.34e+00"
12	“12.0"	“1.2e+01"
120	“120.0"	“1.2e+02"
0.012	“0.012"	“1.2e-02"

Having computed d and k, we now convert the floating-point number to a string based on these values. According to the Schubfach algorithm, d may contain trailing zeros, meaning d mod 10 could be zero. In order to reduce instruction dependencies. We decompose d into two parts: $d//10$ and d mod 10. Let up indicate whether one equals 10, and $updown$ represent the cases where one is either 0 or 10. The following relationships hold:

$$\begin{aligned} d//10 &= m + up \\ updown &= (one = 0 \vee one = 10) = (d \bmod 10 = 0) \\ d &= 10 \cdot (m + up) + (updown?0 : one) \end{aligned} \tag{200}$$

By calculating the approximate range of m, we obtain:

$$\begin{aligned} m &= \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor \\ c \cdot 2^q \cdot 10^{-k-1} &\in [0.1 \cdot c, c) \\ \Rightarrow \lfloor 0.1 \cdot c_{\min} \rfloor \leqslant \lfloor 0.1 \cdot c \rfloor \leqslant m \leqslant \lfloor c \rfloor \leqslant \lfloor c_{\max} \rfloor \end{aligned} \tag{201}$$

Based on the range of c, we derive:

$$\begin{aligned} \text{float} &: \begin{cases} \text{normal} : [\frac{2^{23}+1}{10}, 2^{24} - 1] = [838860.9, 16777215] \\ \text{subnormal} : [\frac{1}{10}, 2^{23} - 1] = [0.1, 8388607] \end{cases} \\ \text{double} &: \begin{cases} \text{normal} : [\frac{2^{52}+1}{10}, 2^{53} - 1] = [4.5 \times 10^{14}, 9 \times 10^{15}] \\ \text{subnormal} : [\frac{1}{10}, 2^{52} - 1] = [0.1, 4.5 \times 10^{15}] \end{cases} \end{aligned} \tag{202}$$

 doi:10.20944/preprints202511.1698.v2

Let $len10(x)$ denote the number of decimal digits in x. For $x > 0$:

$$len10(x) = \lfloor \log_{10}(x) \rfloor + 1 \tag{203}$$

We define $len10(0) = 0$. For example, $len10(12345) = 5$. Consequently:

$$\begin{aligned} \text{float} &: \begin{cases} \text{normal} : len10(m+up) \in [6,8] \\ \text{subnormal} : len10(m+up) \in [0,7] \end{cases} \\ \text{double} &: \begin{cases} \text{normal} : len10(m+up) \in [15,16] \\ \text{subnormal} : len10(m+up) \in [0,16] \end{cases} \end{aligned} \tag{204}$$

Since $d//10 = m + up$, we have $len10(d) = len10(m+up) + 1$. Let $tz10(x)$ denote the number of trailing zeros in the decimal representation of x; for example, $tz10(12300) = 2$. The number of significant digits $sig10(x)$ equals $len10(x) - tz10(x)$. When $updown = 0$, $one \neq 0$, thus $d \bmod 10 = one \neq 0$ and $tz10(d) = 0$. Therefore:

$$\begin{aligned} sig10(d) &= updown?len10(d) - tz10(d) : len10(d) \\ \text{float} &: sig10(d) \in [1,9] \\ \text{double} &: sig10(d) \in [1,17] \end{aligned} \tag{205}$$

For normal floating-point numbers, $len10(d)$ can be computed as:

$$\begin{aligned} \text{float} &: len10(d) = len(m+up) + 1 = 9 - [m+up < 10^7] - [m+up < 10^6] \\ \text{double} &: len10(d) = len(m+up) + 1 = 16 + [(m+up) \geqslant 10^{15}] \end{aligned} \tag{206}$$

where $[P]$ denotes the Iverson bracket, which evaluates to 1 if predicate P is true and 0 otherwise.

Computing the ASCII representation of d reduces to computing the ASCII codes of $m + up$ and one. From the above, we have:

$$\begin{aligned} \text{float} &: m + up < 10^8 \\ \text{double} &: m + up < 10^{16} \end{aligned} \tag{207}$$

These bounds allow efficient computation of the ASCII code for $m + up$ using CPU SIMD instruction sets.

Scalar Implementation

We first present a scalar method that does not rely on SIMD instructions.

Let $x \in [0, 10^8)$ and $dec\_to\_ascii8(x)$ be a function that computes the ASCII representation of x. Algorithm (2) describes this process. This version is an optimized version of the itoa algorithm written by Paul Khuong [18].

Algorithm 2 Convert an 8-digit decimal number to ASCII: dec_to_ascii8(x)

Input: X (type: uint64, range: [0, 10^8))
Output: Y (type: uint64), tz (type: uint)

```
uint64 abcd_efgh = X + (0x100000000 - 10000) * ((X * 0x68db8bbULL) >> 40)
uint64 ab_cd_ef_gh = abcd_efgh + (0x10000 - 100) * (((abcd_efgh * 0x147b) >> 19) & 0x7f0000007f)
uint64 a_b_c_d_e_f_g_h = ab_cd_ef_gh + (0x100 - 10) * (((ab_cd_ef_gh * 0x67) >> 10) & 0xf000f000f000f)
uint tz = __builtin_ctzll(a_b_c_d_e_f_g_h) >> 3
uint64 BCD = CPU_IS_LITTLE_ENDIAN ? byteswap(a_b_c_d_e_f_g_h) : a_b_c_d_e_f_g_h
uint64 Y = BCD + 0x3030303030303030
return Y, tz
```

 doi:10.20944/preprints202511.1698.v2

The function simultaneously computes $tz10(x)$ as the return value tz. Example: X = 12345600, Y="12345600", tz=2.

Algorithm 3 Convert a 16-digit decimal number to ASCII: dec_to_ascii16(x)

Input: X (type: uint64, range: $[0, 10^{16})$)
Output: Y (type: uint128), tz (type: uint)
1: uint64 abcdefgh = X // 10^8
2: uint64 ijklmnop = X mod 10^8
3: (uint64 abcdefgh_ascii, uint tz1) = dec_to_ascii8(abcdefgh)
4: (uint64 ijklmnop_ascii, uint tz2) = dec_to_ascii8(ijklmnop)
5: uint128 Y = CPU_IS_LITTLE_ENDIAN ? (ijklmnop_ascii « 64) + abcdefgh_ascii : (abcdefgh_ascii « 64) + ijklmnop_ascii
6: uint tz = (ijklmnop == 0) ? 8 + tz1 : tz2
7: **return** Y, tz

For $x \in [0, 10^{16})$, let $dec\_to\_ascii16(x)$ compute the ASCII representation of x. Algorithm (3) describes this process. Example: X = 123456001234500, Y="1234560012345600", tz=2.

Handling Undefined Behavior

Since $\_\_builtin\_ctzll(0)$ is undefined behavior in C, special handling is required. For single-precision floating-point numbers, when $m + up = 0$ (the six smallest subnormal numbers), $updown = 0$, avoiding the undefined case. For double-precision numbers, $m + up = 0$ only occurs for 5×10^{-324} (the smallest subnormal number). In our implementation, we handle this special case separately at the function entry.

SIMD Implementation

The SIMD implementations of dec_to_ascii8 and $dec\_to\_ascii16$ follow the same principles as the scalar version. For ARM64 architecture, we use the NEON instruction set; for x64 architecture, we support three variants: AVX512, SSE4.1, and SSE2. Table (4) summarizes these implementations.

Table 4. SIMD implementations of dec_to_ascii8 and $dec\_to\_ascii16$

SIMD implementation	Description
NEON [19]	Original author: Dougall Johnson. Runs on ARM processors with NEON instruction set.
SSE2 [20]	Based on scalar version; requires only SSE2 instruction set.
SSE4.1	Nearly identical to SSE2 implementation; requires SSE4.1 instruction set.
AVX512 [21]	Original author: Daniel Lemire. Requires AVX512IFMA and AVX512VBMI instruction sets.

Using the methods above, we compute the ASCII representation of $m + up$ and the number of significant digits. Printing d is equivalent to printing $m + up$ and one. Based on $sig10(d)$, we determine which buffer contents to retain. In our implementation, we adopt different formats for floating-point numbers in different ranges to enhance readability, as shown in Table (5).

Table 5. Printing formats for different ranges.

Type	Fixed-point	Scientific
float	$[10^{-3}, 10^{7})$	other ranges
double	$[10^{-4}, 10^{16})$	other ranges

In scientific notation, the result includes an exponent. For example, in "1.2e-02", "e-02" represents the exponent. Converting $d \cdot 10^k$ to standard decimal scientific notation yields:

$$d \cdot 10^k = \left(d \cdot 10^{-\lfloor \log_{10}(d) \rfloor}\right) \cdot 10^{k+\lfloor \log_{10}(d) \rfloor} \tag{208}$$

 doi:10.20944/preprints202511.1698.v2

Since $d \cdot 10^{-\lfloor \log_{10}(d) \rfloor} \in [1, 10)$, the exponent is determined by $k + \lfloor \log_{10}(d) \rfloor$. Let $E_{10} = k + \lfloor \log_{10}(d) \rfloor$. Then:

$$E_{10} = k + \lfloor \log_{10}(d) \rfloor = k + \lfloor \log_{10}(m + up) \rfloor + 1 \tag{209}$$

For single-precision floating-point numbers, fixed-point notation is used when $E_{10} \in [-3, 6]$. For double-precision numbers, fixed-point notation is used when $E_{10} \in [-4, 15]$.

The floating-point number printing algorithm proposed in this paper is illustrated as Algorithm 4. The key distinction between our algorithm and others is the use of SIMD instruction sets for converting $m + up$ to ASCII values. For single-precision numbers, we use *dec_to_ascii*8; for double-precision, we use *dec_to_ascii*16. The detailed implementation is available in the source code.

Algorithm 4 Floating-point number printing algorithm

Input: v (type: float/double), buffer (type: char*)
Output: buffer (type: char*)
1: compute $m + up$, $updown$, one, k from v
2: compute $sig10(d) = updown?len10(m + up) - tz10(m + up) : len10(m + up) + 1$
3: compute $E_{10} = k + \lfloor \log_{10}(m + up) \rfloor + 1$
4: convert $m + up$, one to ASCII codes and store in buffer
5: based on $sig10(d)$ and E_{10}, determine which buffer contents to retain
6: if result is in scientific notation, print E_{10}
7: **return** buffer

In our C implementation, all branches are designed as unlikely branches to minimize the impact of branch prediction failures. By transforming the computation of d into computing $m + up$ and one, we reduce instruction dependencies and enhance instruction-level parallelism. Unlike other algorithms that compute the exact value of d before printing, we avoid computing d directly and instead quickly compute $d//10 = m + up$.

3.11. Summary

This chapter explains how to quickly calculate d and k, as well as the printing optimization. Since $d = 10m + one$, the process of calculating d is transformed into calculating m and one.

- Section 3.5 introduces the method for quickly calculating m.
- Section 3.6 and Section 3.7 introduce the methods for quickly calculating one.
- Section 3.9 provides a detailed description of the pseudo-code implementation.
- Section 3.10 discusses print optimization.

Table 6. All algorithms in the benchmark test.

algorithm	float	double	description
Schubfach	Schubfach32	Schubfach64	author:Raffaello Giulietti,https://github.com/c4f7fcce9cb06515/Schubfach.
Schubfach_xjb	Schubfach32_xjb	Schubfach64_xjb	It is improved by Schubfach and has the same output result.
Ryu	Ryu32	Ryu64	author:Ulf Adams,https://github.com/ulfjack/ryu.
Dragonbox	Dragonbox32	Dragonbox64	author:Junekey Jeon,https://github.com/jk-jeon/Dragonbox.
fmt [22]	fmt32	fmt64	author:Victor Zverovich,https://github.com/fmtlib/fmt version:12.1.0
yy_double	-	yy_double	author:GuoYaoYuan,link:yy_double.
yy_json [23]	yy_json32	yy_json64	author:Guo YaoYuan,https://github.com/ibireme/yyjson version:0.12.0
teju_jagua [24]	teju32	teju64	author:Cassio Neri,https://github.com/cassioneri/teju_jagua.
xjb	xjb32	xjb64	this paper,https://github.com/xjb714/xjb.
zmij	zmij32	zmij64	author:Victor Zverovich,https://github.com/vitaut/zmij.
jnum [25]	jnum32	jnum64	author:Jing Leng,https://github.com/lengjingzju/json/jnum.c.
uscalec	-	uscalec	author:Russ Cox,src:https://github.com/rsc/fpfmt commit 6255750 (19 Jan 2026).

 doi:10.20944/preprints202511.1698.v2

4. Experimental Evaluation

This section presents a comprehensive experimental evaluation of the xjb algorithm, comparing its performance against state-of-the-art floating-point to string conversion algorithms across multiple hardware platforms and compilers.

4.1. Correctness Verification

Before conducting performance benchmarks, we verified the correctness of the xjb algorithm through extensive testing:

- **Single-precision (float)**: We tested all possible input values (2^{32} values) and verified that the output matches the Schubfach algorithm, ensuring complete correctness for the binary32 format.
- **Double-precision (double)**: Due to the impracticality of exhaustively testing all 2^{64} possible values, we conducted comprehensive random testing with a large sample size to verify correctness for the binary64 format.

All test results confirmed that xjb produces output identical to Schubfach while satisfying the SW principle.

4.2. Experimental Setup

4.2.1. Hardware Platforms

We conducted benchmarks on two representative hardware platforms:

- **AMD Ryzen 7 7840H**: A modern x86-64 processor with support for AVX2 and AVX-512 instruction sets, running Ubuntu 26.04.
- **Apple M1**: A representative ARM-based processor with NEON SIMD support, running macOS 26.4.

4.2.2. Compilers and Compilation Flags

- **AMD R7-7840H**: Intel C++ Compiler (icpx) version 2025.0.4
- **Apple M1**: Apple Clang version 21.0.0

All benchmarks were compiled with the optimization flags `-O3 -march=native` to enable maximum compiler optimizations and architecture-specific instruction generation.

4.2.3. Benchmark Methodology

The benchmark methodology was designed to provide fair and reproducible performance measurements:

1. Generate 2^{24} random floating-point numbers, excluding special values (0, NaN, and Inf).
2. Measure the total time required to convert all numbers to decimal representation.
3. Calculate the average conversion time per floating-point number.

This methodology ensures statistical significance while avoiding the overhead of special-case handling that would skew results.

4.3. Algorithms Compared

We compared xjb against the following state-of-the-art algorithms:

Tables 7 and (8) present the benchmark results for float-to-decimal and double-to-decimal conversions on AMD R7-7840H and Apple M1, respectively. Figure 1a–f presents the benchmark results for the conversion from float to string and from double to string. Figures 1a–d present the benchmark results when the input values are completely random. Since the significant digit length range of the printing results is from 1 to 17, comparative tests were also conducted for different effective lengths, as shown in Figures 1e,f.

- **Schubfach**: The baseline algorithm from which xjb is derived.

 doi:10.20944/preprints202511.1698.v2

- **Schubfach_xjb**: A variant of Schubfach with xjb optimizations.
- **Ryu**: A widely-used high-performance algorithm.
- **Dragonbox**: Another state-of-the-art algorithm with competitive performance.
- **fmt**: A modern formatting library.
- **yy_json / yy_double**: Fast algorithms optimized for JSON serialization.
- **teju_jagua**: A recent algorithm (note: only supports float/double to decimal conversion).
- **zmij**: A modern algorithm with competitive performance.
- **uscalec**: An algorithm supporting double-precision only.

4.4. Performance Results

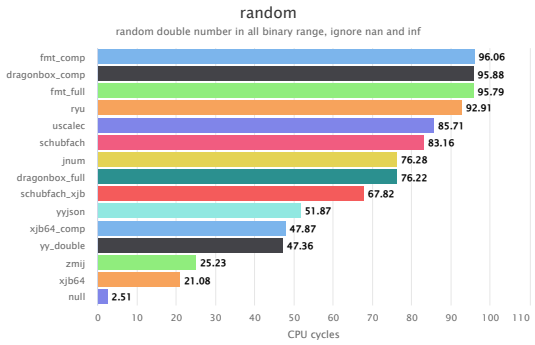
Table 7. Float/double to decimal benchmark results on AMD R7-7840H and ubuntu 26.04. The unit is nanosecond(ns).

algorithm	float	double
	icpx 2025.0.4	icpx 2025.0.4
Schubfach	12.2	11.51
Schubfach_xjb	4.44	6.33
Ryu	14.02	13.08
Dragonbox	10.19	10.05
yy_json	4.67	5.72
yy_double	-	5.24
teju_jagua	14.99	14.37
zmij	4.76	4.78
uscalec	-	11.27
xjb	2.24	3.76

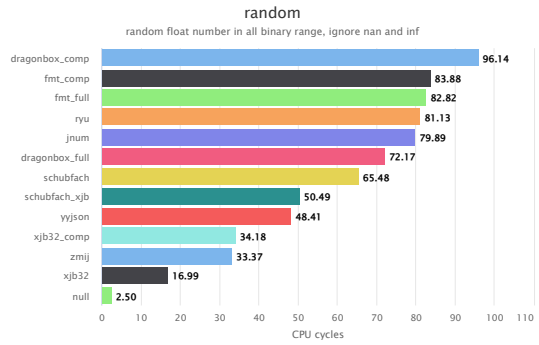
Table 8. Float/double to decimal benchmark results on Apple M1 and MacOS 26.4. The unit is nanosecond(ns).

algorithm	float	double
	apple clang 21.0.0	apple clang 21.0.0
Schubfach	11.64	13.12
Schubfach_xjb	5.16	6.58
Ryu	15.75	14.16
Dragonbox	11.78	12.03
yy_json	3.97	4.46
yy_double	-	4.08
teju_jagua	20.25	18.66
zmij	4.11	3.83
uscalec	-	15.26
xjb	2.15	2.58

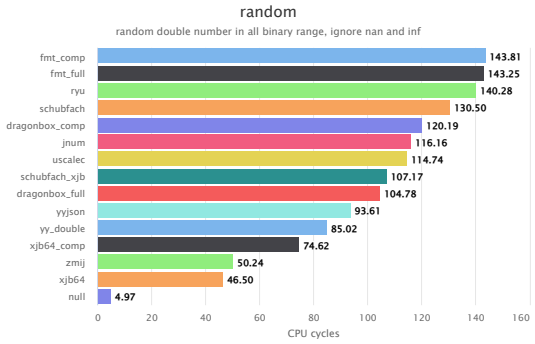
 doi:10.20944/preprints202511.1698.v2



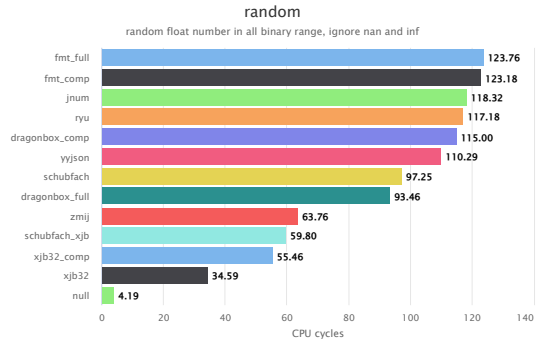
(a) Apple M1 random double benchmark result



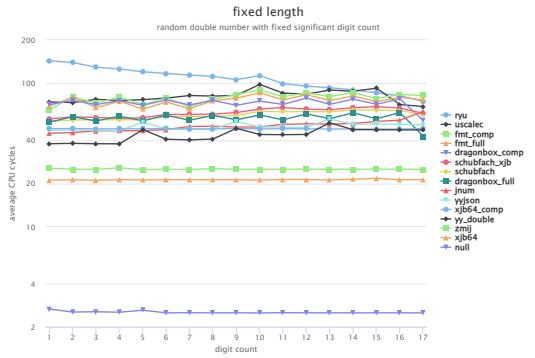
(b) Apple M1 random float benchmark result



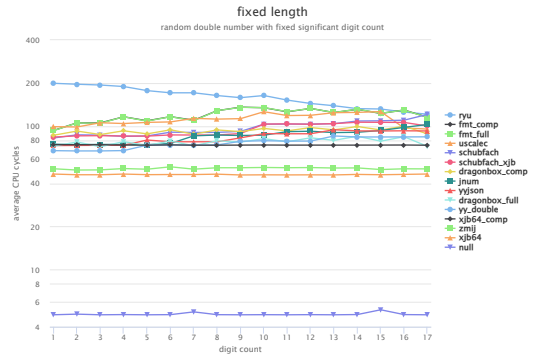
(c) AMD R7-7840H random double benchmark result



(d) AMD R7-7840H random float benchmark result



(e) Apple M1 fixed-length double benchmark result



(f) AMD R7-7840H fixed-length double benchmark result

Figure 1. float/double to string benchmark results.

4.5. Analysis and Discussion

4.5.1. Performance Comparison

The benchmark results demonstrate that xjb achieves superior performance compared to all other algorithms across both hardware platforms:

On AMD R7-7840H:

- For float-to-decimal conversion, xjb achieves 2.24 ns, which is 1.98× faster than the second-fastest algorithm (Schubfach_xjb at 4.44 ns) and 5.44× faster than the baseline Schubfach (12.2 ns).
- For double-to-decimal conversion, xjb achieves 3.76 ns, which is 1.27× faster than the second-fastest algorithm (yy_double at 5.24 ns) and 3.06× faster than the baseline Schubfach (11.51 ns).
- For float-to-string conversion, xjb is about 60% faster than zmij.
- For double-to-string conversion, xjb is about 8% faster than zmij.

On Apple M1:

- For float-to-decimal conversion, xjb achieves 2.15 ns, which is 1.84× faster than the second-fastest algorithm (zmij at 4.11 ns) and 5.41× faster than the baseline Schubfach (11.64 ns).

 doi:10.20944/preprints202511.1698.v2

- For double-to-decimal conversion, xjb achieves 2.58 ns, which is 1.49× faster than the second-fastest algorithm (yy_double at 4.08 ns) and 5.08× faster than the baseline Schubfach (13.12 ns).
- For float-to-string conversion, xjb is about 96% faster than zmij.
- For double-to-string conversion, xjb is about 20% faster than zmij.

4.5.2. Performance Consistency

A key advantage of the xjb algorithm is its consistent performance across different input distributions. Since all branches in the xjb implementation are designed to be unlikely branches, the algorithm maintains a high branch prediction success rate regardless of input patterns. This property is particularly valuable in real-world applications where input distributions may vary unpredictably.

4.5.3. Cross-Platform Performance

The benchmark results demonstrate that xjb performs well across different processor architectures:

- On x86-64 (AMD R7-7840H), xjb benefits from the compiler's ability to generate efficient code for the algorithm's arithmetic operations.
- On ARM64 (Apple M1), xjb maintains its performance advantage, demonstrating the algorithm's portability and effectiveness across different instruction set architectures.

4.5.4. Comparison with Related Algorithms

- **vs. Schubfach**: xjb achieves 3–5× speedup over the baseline Schubfach algorithm, validating the effectiveness of our optimizations in reducing multiplication operations and minimizing branch prediction penalties.
- **vs. yy_json/yy_double**: While these algorithms are highly optimized for JSON serialization workloads, xjb outperforms them by 1.3–1.8×, demonstrating broader applicability beyond specific use cases.
- **vs. zmij**: xjb achieves 1.5–1.9× speedup over zmij, another recent high-performance algorithm, showing the benefits of our approach to instruction dependency reduction.
- **vs. Ryu and Dragonbox**: xjb significantly outperforms these widely-used algorithms (2.5–6× faster), demonstrating that the optimizations in xjb provide substantial benefits over established approaches.

4.6. Summary

The experimental evaluation demonstrates that xjb achieves state-of-the-art performance for floating-point to decimal conversion across multiple hardware platforms. The algorithm's key strengths include:

1. Consistently superior performance across both x86-64 and ARM64 architectures.
2. Significant speedups over the baseline Schubfach algorithm (3–5×).
3. Competitive or superior performance compared to all other state-of-the-art algorithms.
4. Stable performance regardless of input distribution due to effective branch prediction optimization.

These results validate the effectiveness of our optimization strategies: reducing multiplication operations, minimizing instruction dependencies, and designing branch patterns that work well with modern processor pipelines.

5. Conclusions

This paper presented xjb, a novel high-performance algorithm for converting IEEE 754 floating-point numbers to decimal string representations. Building upon the Schubfach algorithm, xjb introduces several key optimizations that significantly improve conversion speed while maintaining full compliance with the Steele-White principle for accurate and minimal-length output.

 doi:10.20944/preprints202511.1698.v2

5.1. Summary of Contributions

The main contributions of this work are:

1. **Reduced computational complexity**: By restructuring the calculation process and minimizing the number of high-precision multiplication operations, xjb achieves substantial performance improvements over the baseline Schubfach algorithm.
2. **Optimized branch structure**: The algorithm employs unlikely branch patterns that enable efficient branch prediction on modern processors, resulting in consistent performance across diverse input distributions.
3. **Cross-platform portability**: xjb demonstrates excellent performance across both x86-64 and ARM64 architectures, making it suitable for deployment in heterogeneous computing environments.
4. **Comprehensive validation**: The algorithm has been thoroughly tested and verified to produce correct output for all IEEE 754 binary32 and binary64 floating-point values.

5.2. Experimental Findings

Our experimental evaluation on AMD R7-7840H and Apple M1 processors demonstrates that xjb achieves state-of-the-art performance:

- 3–5× speedup over the baseline Schubfach algorithm
- Faster than other high-performance algorithms such as yy_json, yy_double, and zmij
- Consistent performance across different input patterns and hardware platforms

5.3. Practical Implications

The xjb algorithm has immediate practical applications in numerous domains:

- **Data serialization**: JSON and other text-based data formats require efficient floating-point to string conversion for serialization operations.
- **Scientific computing**: Applications that output numerical results in human-readable format benefit from faster conversion without sacrificing accuracy.
- **Database systems**: Export operations and query result formatting can leverage xjb for improved throughput.
- **Web services**: RESTful APIs and web applications that return numerical data can achieve lower latency with efficient conversion.

5.4. Limitations and Future Work

While xjb demonstrates strong performance, several areas warrant further investigation:

1. **Extended precision support**: Future work could extend xjb to support extended precision formats (e.g., 80-bit and 128-bit floating-point numbers) for applications requiring higher precision.
2. **SIMD vectorization**: Although xjb is designed to be SIMD-friendly, explicit vectorization using AVX-512 or NEON could yield additional performance gains for batch conversion workloads.
3. **Compiler compatibility**: Further optimization for different compilers (particularly MSVC) would improve portability across development environments.
4. **Memory-constrained environments**: Investigating memory-efficient variants of xjb could benefit embedded systems and other resource-constrained platforms.

5.5. Availability

The complete implementation of the xjb algorithm, along with benchmark tools and test suites, is publicly available at https://github.com/xjb714/xjb/releases/tag/v1.0.0. We encourage the community to integrate, test, and contribute to the ongoing development of this work.

 doi:10.20944/preprints202511.1698.v2

References

1. G. L. Steel Jr. and J. L. White. How to Print Floating-Point Numbers Accurately. *In Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, PLDI 1990.* ACM, New York, NY, USA, 112-126. https://doi.org/10.1145/93542.93559
2. G. L. Steel Jr. and J. L. White. How to Print Floating-Point Numbers Accurately (Retrospective). ACM SIGPLAN Notices 39(4), April 2004 (Best of PLDI, 1979-1999). https://dl.acm.org/doi/10.1145/989393.989431
3. Robert G. Burger and R. Kent Dybvig. 1996. Printing Floating-point Numbers Quickly and Accurately. In Proceedings of the ACM SIGPLAN1996 Conference on Programming Language Design and Implementation (PLDI '96). ACM, New York, NY, USA, 108ś116. https://doi.org/10.1145/231379.231397
4. F. Loitsch. Printing Floating-Point Numbers Quickly and Accurately with Integers. *In Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation, PLDI 2010.* ACM, New York, NY, USA, 233-243. https://doi.org/10.1145/1806596.1806623
5. M. Andrysco, R. Jhala, and S. Lerner. Printing Floating-Point Numbers: a Faster, Always Correct Method. *In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016.* ACM, New York, NY, USA, 555-567. https://doi.org/10.1145/2837614.2837654
6. Ulf Adams. 2018. Ryū: Fast Float-to-String Conversion. *In Proceed- ings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3192366.3192369
7. Ulf Adams. 2019. Ryū Revisited: Printf Floating Point Conversion. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 169 (October 2019), 23 pages. https://doi.org/10.1145/3360595
8. R. Giulietti. The Schubfach Way to Render Doubles. 2020. https://drive.google.com/file/d/1KLtG_LaIbK9ETXI290zqCxvBW94dj058/view (Sep. 2020)
9. J. Jeon. Grisu-Exact: A Fast and Exact Floating-Point Printing Algorithm. 2020. https://github.com/jk-jeon/Grisu-Exact/blob/master/other_files/Grisu-Exact.pdf. (Sep. 2020)
10. Junekey Jeon. 2024. Dragonbox: A New Floating-Point Binary-to-Decimal Conversion Algorithm. https://github.com/jk-jeon/Dragonbox
11. Guo YaoYuan. https://github.com/ibireme/c_numconv_benchmark/blob/master/vendor/yy_double/yy_double.c (Nov. 2024)
12. Russ Cox. https://github.com/rsc/fpfmt (Jan. 2026)
13. Russ Cox. Floating-Point Printing and Parsing Can Be Simple And Fast. published online, https://research.swtch.com/fp(Jan. 2026)
14. Russ Cox. Fast Unrounded Scaling: Proof by Ivy. published online, https://research.swtch.com/fp-proof(Jan. 2026)
15. Victor Zverovich. https://github.com/vitaut/zmij (Mar. 2026)
16. IEEE Standard for Binary Floating-Point Arithmetic, in ANSI/IEEE Std 754-1985 , vol., no., pp.1-20, 12 Oct. 1985. https://doi.org/10.1109/IEEESTD.1985.82928
17. IEEE Standard for Floating-Point Arithmetic, in IEEE Std 754-2019 (Revision of IEEE 754-2008) , vol., no., pp.1-84, 22 July 2019, https://doi.org/10.1109/IEEESTD.2019.8766229
18. Paul Khuong. How to print integers really fast (with Open Source AppNexus code!).published online, https://pvk.ca/Blog/2017/12/22/appnexus-common-framework-its-out-also-how-to-print-integers-faster/.(Dec. 2017)
19. Dougall Johnson. Converting integers to fixed-width strings faster with Neon SIMD on the Apple M1, published online, https://dougallj.wordpress.com/2022/04/01/converting-integers-to-fixed-width-strings-faster-with-neon-simd-on-the-apple-m1/ (Apr. 2022)
20. Wojciech Muła. SSE: conversion integers to decimal representation. published online, http://0x80.pl/notesen/2011-10-21-sse-itoa.html. (Oct. 2011)
21. Daniel Lemire. Converting integers to decimal strings faster with AVX-512, published online, https://lemire.me/blog/2022/03/28/converting-integers-to-decimal-strings-faster-with-avx-512/ (Mar. 2022)
22. Victor Zverovich. https://github.com/fmtlib/fmt (Oct. 2025)
23. Guo YaoYuan. https://github.com/ibireme/yyjson (Aug. 2025)
24. Cassio Neri. https://github.com/cassioneri/teju_jagua (Nov. 2025)
25. Jing Leng. https://github.com/lengjingzju/json/jnum.c (Nov. 2025)

 doi:10.20944/preprints202511.1698.v2