High-Performance Decimal Floating-Point Arithmetic: Algorithms and Implementation in C++

MATTHEW BORLAND and CHRISTOPHER KORMANYOS

This article presents a comprehensive, portable C++ implementation of decimal floating-point arithmetic conforming to IEEE 754-2019 standards[8]. Our system offers three IEEE 754-compliant types, and three additional types that prioritize performance over strict standard adherence, providing flexible options for various computational needs. We describe in detail the Decimal system architecture, its standard library, and usage guidelines. The implementation incorporates novel algorithms for key operations, significantly improving performance in common use cases. Rigorous testing results demonstrate the system's correctness and IEEE 754 compliance. Performance benchmarks show competitive or superior results compared to existing implementations. This work addresses the growing demand for precise decimal arithmetic in financial, scientific, and engineering applications, offering a robust, efficient solution for C++ developers.

CCS Concepts: • **General and reference** → **Design**; **Performance**; • **Mathematics of computing** → **Mathematical software performance**; • **Software and its engineering** → *Software architectures.*

Additional Key Words and Phrases: C++, Decimal Floating Point, Object Oriented, Special Functions, System Architecture

ACM Reference Format:
Matthew Borland and Christopher Kormanyos. 2018. High-Performance Decimal Floating-Point Arithmetic: Algorithms and Implementation in C++. *ACM Trans. Math. Softw.* 37, 4, Article 111 (August 2018), 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Decimal floating-point arithmetic is crucial for many applications[4], particularly in financial and scientific computing. While several C++ decimal floating-point packages exist, they often lack IEEE 754[8] conformance, interoperability with the C++ Standard Template Library (STL), or both, and have limited portability. This paper presents a novel decimal system that addresses these limitations and advances decimal floating-point technology. Our system is standalone, relies on a minimal subset of the C++ STL, and has been tested on a wide range of devices, from S390X mainframes to AVR boards. It provides seamless interoperability with existing C++ standard types and full IEEE 754 conformance, features not collectively offered by any known package. This work significantly enhances the toolset available for high-precision decimal computations across diverse computing environments.

2 The Decimal System

2.1 Background

Decimal floating point is a method of representing floating point numbers using base-10 instead of base-2 like most are familiar with. Decimal floating point is not a new concept per se. Mechanical

Authors' Contact Information: Matthew Borland, matt@mattborland.com; Christopher Kormanyos, e_float@yahoo.com.

ACM 1557-7295/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

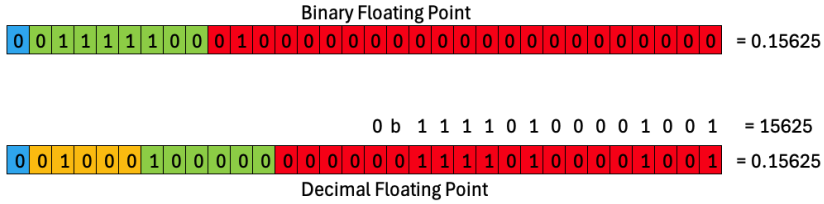

Fig. 1. Bit layouts of Binary and Decimal Floating Point Numbers

devices such as the abacus and slide rule use base-10 numbers. Early computers such as the IBM 650 used decimal floating point numbers prior to IEEE standardization[2]. Some modern architectures support decimal floating point in hardware such as IBM System Z[15]. The advantage of decimal floating point is that it can represent human readable floating point numbers exactly unlike binary floating point.

The major difference between the layout of binary and decimal floating-point numbers lies in their internal structure and how they represent the same numerical value. Figure 1 illustrates this difference by showing the bit layout of the same number in both formats.

Binary floating-point numbers consist of three parts:

- The sign bit (shown in blue)
- The exponent (shown in green)
- The significand (shown in red)

Decimal floating-point numbers, on the other hand, are composed of four distinct parts:

- The sign bit (shown in blue)
- The combination field (shown in orange)
- The exponent continuation (shown in green)
- The coefficient continuation (shown in red)

As depicted in Figure 1, these different structures allow the same numerical value to be represented in two fundamentally different ways. The binary format uses a base-2 system, while the decimal format uses a base-10 system, which affects how precisely certain decimal values can be represented and how rounding errors propagate in calculations. Due to these differences decimal can represent numbers exactly that binary cannot, for example the number 0.3 which with a C++ `double` would actually be: 0.30000000000000004. There are actually two different bit layouts for decimal floating point types: Binary Integer Significand Field (BID), and Densely Packed Decimal Significand Field (DPD)[8]. The above example is depicted in the DPD format which is typically for hardware implementation. For the remainder of this paper we will focus on the BID format as it is more natural for software implementations like ours.

2.1.1 Cohorts. Another idiosyncrasy that is found in decimal floating point types are called cohorts[8]. A cohort is all the different ways to represent the same value. For example one can represent the number 10 as: 0.1×10^2, 1×10^1, 10×10^0, 100×10^{-1}, and so on. The ramification of this is we must normalize the significand and exponents of a number before we can attempt doing comparisons or other basic mathematical operations.

2.2 Provided Types

The decimal system provides 6 total types: `decimal32_t`, `decimal64_t`, `decimal128_t`, `decimal_fast32_t`, `decimal_fast64_t`, and `decimal_fast128_t`. The first three types are conformant to IEEE-754

standards while the latter three provided identical numerical results without the constraints of IEEE-754.

The three conformant types is the design specifications provided in IEEE-754 namely their properties:

Parameter	decimal32	decimal64	decimal128
Storage Width	32	64	128
Precision (decimal digits)	7	16	34
Max Exponent	96	384	6144
Exponent Bias	101	398	6176
Sign Width	1	1	1
Combination Field Width	11	13	17
Significand Continuation Width	20	50	110

Table 1. Decimal Floating-Point Format Parameters

The three so-called fast types (i.e. `decimal_fastXX_t`) have the same values of precision, range, and exponent as their analogous conformant type, but require more space. This is a similar approach to `std::uint_fastXX_t` types which have at least the same values as the standard types.

2.3 System Architecture

The decimal system architecture is robust and flexible. Each type is implemented completely independently of each other, but they share many of the same function implementations from the STL.

2.4 Using the System

Every effort was made during design and implementation to ensure that using the decimal system was straightforward and intuitive. The library contains zero dependencies, and uses only a small subsection of the C++ STL. The required language standard for the library is C++14.

```
1  #include <boost/decimal.hpp>
2  #include <iostream>
3  #include <iomanip>
4
5  int main()
6  {
7      using namespace boost::decimal;
8
9      constexpr decimal32 val_1 {100};         // Construction from an
       integer
10     constexpr decimal32 val_2 {10, 1};       // Construction from an
       integer and exponent
11     constexpr decimal32 val_3 {1U, 2, false}; // Construction from an
       integer, exponent, and sign
12
13     std::cout << "Val_1: " << val_1 << '\n'
14               << "Val_2: " << val_2 << '\n'
15               << "Val_3: " << val_3 << '\n';
```

```
16
17      if (val_1 == val_2 && val_2 == val_3 && val_1 == val_3)
18      {
19          std::cout << "All equal values" << std::endl;
20      }
21
22      constexpr decimal64 val_4 {decimal64{2, -1} + decimal64{1, -1}};
23      constexpr double float_val_4 {0.2 + 0.1};
24      const decimal64 val_5 { float_val_4 }; // Explicit Conversion from
        double
25
26      std::cout << std::setprecision(17) << "Val_4: " << val_4 << '\n'
27                <<  "Float: " << float_val_4 << '\n'
28                <<  "Val_5: " << val_5 << '\n';
29
30      if (val_4 == val_5)
31      {
32          std::cout << "Floats are equal" << std::endl;
33      }
34      else
35      {
36          std::cout << "Floats are not equal" << std::endl;
37      }
38
39      return 0;
40  }
```

Listing 1. Basic Usage

In Listing 1 we show how to construct the type and that it works just like a built-in flowing point type. The following is an example that leverages one of decimal-floating points main strengths and one that we expect many people will use.

```
1  #include <boost/decimal.hpp>
2  #include <iostream>
3  #include <cassert>
4
5  int main()
6  {
7      using namespace boost::decimal;
8
9      decimal64 val {0.25}; // Construction from a double (not recommended
       but explicit construction is allowed)
10
11     char buffer[256];
12     auto r_to = to_chars(buffer, buffer + sizeof(buffer) - 1, val);
13     assert(r_to); // checks std::errc()
14     *r_to.ptr = '\0';
15
16     decimal64 return_value;
```

```
    auto r_from = from_chars(buffer, buffer + std::strlen(buffer),
    return_value);
    assert(r_from);

    assert(val == return_value);

    std::cout << " Initial Value: " << val << '\n'
              << "Returned Value: " << return_value << std::endl;

    return 0;
}
```

Listing 2. Basic Usage

In Listing 2 we show how to serialize and parse numbers. These techniques and functions are an extension of Boost.Charconv[3].

3 Implementation Details

3.1 IEEE 754 Conformant Decimal Types

The decimal system provides three IEEE-754 compliant types: `decimal32`, `decimal64`, and `decimal128`. Each of these are constructed using a single unsigned integer of the same width to hold the bits. For example `decimal32` consists of a single `std::uint32_t`. `decimal128` uses a custom implementation of a 128-bit unsigned integer for portability reasons rather than relying on the existence of `unsigned __int128`. The discussion of implementing big integers is outside the scope of the paper, but papers and implementations can be found in numerous places[3][12][10].

The decoding step of `decimal32` is somewhat involved because the values of the significand and exponent depend on the bits in the combination field. Our first case is where the bits in the combination field are in the form `00XXX`, `01XXX`, and `10XXX`.

```
//    Comb.  Exponent          Significand
// s         eeeeeeee     ttttttttttttttttttttttt
// s   11    eeeeeeee    [100]ttttttttttttttttttttt
```

Listing 3. Combination Field Pattern 1

In Listing 3 we can see that with these combination field bit patterns that most of the time a decimal floating point number is actually in the same bit layout as a binary floating point number. The special case is when we need more than 23 bits to represent the significand of our number. In this case we use 2 bits in the combination field to imply 3 bits in the significand which gives us the full range of significand values.

The final case is where the bits in the combination field are used for non-finite numbers:

```
static constexpr std::uint32_t d32_inf_mask  = UINT32_C(0x78000000);
static constexpr std::uint32_t d32_nan_mask  = UINT32_C(0x7C000000);
static constexpr std::uint32_t d32_snan_mask = UINT32_C(0x7E000000);
```

Listing 4. Combination Field Pattern 3

Since this combination field pattern is only for non-finite numbers it makes the implementation of `isnan`, `isinf`, etc. straightforward. We can also see the signbit remains unused so we can represent signed infinities. One can also use the remaining exponent and significand bits to provide a payload to NaN analogous to the use of `std::nan`[6].

Now that we have seen what each bit pattern of the combination field corresponds to; the full implementation of decoding the significand then becomes:

```
constexpr auto decimal32_t::full_significand() const noexcept ->
    significand_type
{
    significand_type significand {};

    if ((bits_ & detail::d32_comb_11_mask) == detail::d32_comb_11_mask)
    {
        constexpr std::uint32_t implied_bit {UINT32_C(0
    b10000000000000000000000)};
        significand = implied_bit | (bits_ & detail::
    d32_11_significand_mask);
    }
    else
    {
        significand = bits_ & detail::d32_not_11_significand_mask;
    }

    return significand;
}
}
```

Listing 5. Decoding decimal32_t significand

Encoding the value is a similar to the decoding process found in Listing 5 , but more complex process in that we need to work backwards through the steps to figure out what the value of the combination field needs to be, and then we use a series of masks to write the value.

3.2 IEEE 754 Fast Types

Now that we have discussed the three IEEE-754 conformant types we will turn our attention to much more performant, but non-compliant types. The library offers: `decimal32_fast`, `decimal64_fast`, and `decimal128_fast`. These types are similar to how instead of `std::uint32_t` you can use `std::uint_fast32_t` which on x86_64 platforms generally aliases to `std::uint64_t`. This is the classic optimization of trading space for time. Again we will focus on the 32-bit type for clarity and simplicity. Unlike `decimal32` consisting of a data `std::uint32_t`, `decimal32_fast` actually contains its internal state in a structure.

```
class decimal32_fast final
{
public:
    using significand_type = std::uint_fast32_t;
    using exponent_type = std::uint_fast8_t;
    using biased_exponent_type = std::int_fast32_t;

private:
    // In regular decimal32 we have to decode the 24 bits of the
    significand and the 8 bits of the exp
```

```
10      // Here we just use them directly at the cost of at least 2 extra
        bytes of internal state
11      // since the fast integer types will be at least 32 and 8 bits
        respectively
12
13      significand_type significand_ {};
14      exponent_type exponent_ {};
15      bool sign_ {};
16
17      // Continued
18  };
```

Listing 6. Internal state of decimal32_fast

In Listing 6 we see that now instead of having to encode and decode the number every time we perform an operation we can now just access the value directly:

```
1  constexpr auto full_significand() const noexcept -> significand_type
2  {
3      return significand_;
4  }
```

Listing 7. Decoding decimal32_fast significand

Now compare the decoding process found in the conformant type in Listing 5 to that in Listing 7. Encoding the value is similarly trivial. Each of the fast types use the same approach, but with different types used for the internal state to ensure that the range and precision is equal to that of the IEEE 754 conformant types.

An additional optimization for the fast types is that exponent and significand are stored in normalized form. As discussed in Section 2.1.1 we would have to normalize the values of the exponent and significand prior to any comparison or mathematical operation. Rather than having to normalize each time we perform an operation we do it exactly one time, when the number is constructed. This is beneficial because normalization has time complexity of $O(log(n))$ whereas most basic operations are only $O(1)$.

```
1  // Converts the significand to full precision to remove the effects of
       cohorts
2  template <typename TargetDecimalType, typename T1, typename T2>
3  constexpr auto normalize(T1& significand, T2& exp, bool sign = false)
       noexcept -> void
4  {
5      constexpr auto target_precision {detail::precision_v<
       TargetDecimalType>};
6      const auto digits {num_digits(significand)};
7
8      if (digits < target_precision)
9      {
10         const auto zeros_needed {target_precision - digits};
11         significand *= pow10(static_cast<T1>(zeros_needed));
12         exp -= zeros_needed;
13     }
14     else if (digits > target_precision)
```

```
    {
        const auto excess_digits {digits - (target_precision + 1)};
        significand /= pow10(static_cast<T1>(excess_digits));
        // Perform final rounding according to the fenv rounding mode
        exp += detail::fenv_round<TargetDecimalType>(significand, sign ||
     significand < 0U) + excess_digits;
    }
}
```

Listing 8. Normalization Method

As we can see in the above listing in the event that our significand has too few digits we append a fixed number of zeros, and if it's too many (e.g. in the constructor) we must remove excess and round appropriately.

3.3 Basic Operations

Now that we have discussed how to encode and decode the sign, exponent, and significand of a decimal type number we will cover basic operations.

3.3.1 Comparisons. All types provided by this system support >, >=, ! =, ==, <=, <, and when using C++20 <=>. As discussed in Section 2.1.1 we will need to normalize the values of the exponent and significand to ensure fair comparisons. Once we have normalized our values using the methods from Listing 8 we are then able to implement these operations by comparing the values of the significand, exponent, and sign. We must also check for non-finite values such as NaN which aren't comparable values. For an example the implementation of equality is depicted below:

```
template <BOOST_DECIMAL_DECIMAL_FLOATING_TYPE DecimalType>
BOOST_DECIMAL_FORCE_INLINE constexpr auto equality_impl(DecimalType lhs,
    DecimalType rhs) noexcept -> bool
{
    using comp_type = typename DecimalType::significand_type;

    // Step 1: Check for NANs per IEEE 754
    #ifndef BOOST_DECIMAL_FAST_MATH
    if (isnan(lhs) || isnan(rhs))
    {
        return false;
    }
    #endif

    // Step 2: Fast path
    if (lhs.bits_ == rhs.bits_)
    {
        return true;
    }

    const auto lhs_components {lhs.to_components()};
    const auto rhs_components {rhs.to_components()};

    auto lhs_sig {lhs_components.sig};
```

```
24      auto rhs_sig {rhs_components.sig};
25
26      // Step 4: Check signs
27      if (lhs_components.sign != rhs_components.sign)
28      {
29          return (lhs_sig == 0U && rhs_sig == 0U);
30      }
31
32      // Step 5: Check the exponents
33      // If the difference is greater than we can represent in the
        significand than we can assume they are different
34      const auto lhs_exp {lhs_components.exp};
35      const auto rhs_exp {rhs_components.exp};
36
37      const auto delta_exp {lhs_exp - rhs_exp};
38
39      if (delta_exp > detail::precision_v<DecimalType> || delta_exp < -
        detail::precision_v<DecimalType>)
40      {
41          return false;
42      }
43
44      // Step 6: Normalize the significand and compare
45      using promoted_sig_type = std::conditional_t<std::is_same<typename
        DecimalType::significand_type, std::uint32_t>::value, std::uint64_t,
        int128::uint128_t>;
46      promoted_sig_type promotes_lhs {lhs_sig};
47      promoted_sig_type promotes_rhs {rhs_sig};
48
49      if (delta_exp > 0)
50      {
51          promotes_lhs *= detail::pow10(static_cast<promoted_sig_type>(
        delta_exp));
52      }
53      else if (delta_exp < 0)
54      {
55          promotes_rhs *= detail::pow10(static_cast<promoted_sig_type>(-
        delta_exp));
56      }
57
58       return promotes_lhs == promotes_rhs;
59  }
```

Listing 9. Equality Method

3.3.2 Add and Sub. We are able to utilize the property that $a + b == a - (-b)$, as well as commutativity of operations ($a + b == b + a$) to unify our implementations of addition and subtraction.

Fundamentally the implementation of addition is straightforward, except for the handling of differing exponents. If we have $a + b$ where normalized a has a larger exponent than normalized b we multiply a by $10^{\Delta exponents}$ This allows us to maintain the full precision for rounding unlike if we divided by the same power of ten. Once we have accounted for the difference in exponents, we add these modified significands, and construct a new decimal number with the result. In it's complete form 32-bit addition is shown below:

```
template <typename ReturnType, typename T>
constexpr auto d32_add_impl(const T& lhs, const T& rhs) noexcept ->
    ReturnType
{
    // Each of the significands are maximally 23 bits.
    // Rather than doing division to get proper alignment we will promote
     to 64 bits
    // And do a single mul followed by an add
    using add_type = std::int_fast64_t;
    using promoted_sig_type = std::uint_fast64_t;

    promoted_sig_type big_lhs {lhs.full_significand()};
    promoted_sig_type big_rhs {rhs.full_significand()};
    auto lhs_exp {lhs.biased_exponent()};
    const auto rhs_exp {rhs.biased_exponent()};

    // Align to larger exponent
    if (lhs_exp != rhs_exp)
    {
        constexpr auto max_shift {detail::make_positive_unsigned(std::
    numeric_limits<promoted_sig_type>::digits10 - detail::precision_v<
    decimal32_t> - 1)};
        const auto shift {detail::make_positive_unsigned(lhs_exp -
    rhs_exp)};

        if (shift > max_shift)
        {
            auto round {rounding_mode::fe_dec_default};

            if (!BOOST_DECIMAL_IS_CONSTANT_EVALUATED(lhs))
            {
                round = fegetround();
            }

            if (BOOST_DECIMAL_LIKELY(round != rounding_mode::
    fe_dec_downward && round != rounding_mode::fe_dec_upward))
            {
                return big_lhs != 0U && (lhs_exp > rhs_exp) ?
                                    ReturnType{lhs.full_significand(),
    lhs.biased_exponent(), lhs.isneg()} :
```

```
34                                      ReturnType{rhs.full_significand(),
    rhs.biased_exponent(), rhs.isneg()};
35              }
36              else if (round == rounding_mode::fe_dec_downward)
37              {
38                  // If we are subtracting even disparate numbers we need
    to round down
39                  // E.g. "5e+95"_DF - "4e-100"_DF == "4.999999e+95"_DF
40
41                  using sig_type = typename T::significand_type;
42
43                  return big_lhs != 0U && (lhs_exp > rhs_exp) ?
44                      ReturnType{lhs.full_significand() - static_cast<
    sig_type>(lhs.isneg() != rhs.isneg()), lhs.biased_exponent(), lhs.
    isneg()} :
45                      ReturnType{rhs.full_significand() - static_cast<
    sig_type>(lhs.isneg() != rhs.isneg()), rhs.biased_exponent(), rhs.
    isneg()};
46              }
47              else
48              {
49                  // rounding mode == fe_dec_upward
50                  // Unconditionally round up. Could be 5e+95 + 4e-100 ->
    5.000001e+95
51                  return big_lhs != 0U && (lhs_exp > rhs_exp) ?
52                      ReturnType{lhs.full_significand() + 1U, lhs.
    biased_exponent(), lhs.isneg()} :
53                      ReturnType{rhs.full_significand() + 1U, rhs.
    biased_exponent(), rhs.isneg()};
54              }
55          }
56
57          if (lhs_exp < rhs_exp)
58          {
59              big_rhs *= detail::pow10<promoted_sig_type>(shift);
60              lhs_exp = rhs_exp - static_cast<decimal32_t_components::
    biased_exponent_type>(shift);
61          }
62          else
63          {
64              big_lhs *= detail::pow10<promoted_sig_type>(shift);
65              lhs_exp -= static_cast<decimal32_t_components::
    biased_exponent_type>(shift);
66          }
67      }
68
69      // Perform signed addition with overflow protection
```

```
70      const auto signed_lhs {detail::make_signed_value<add_type>(
        static_cast<add_type>(big_lhs), lhs.isneg())};
71      const auto signed_rhs {detail::make_signed_value<add_type>(
        static_cast<add_type>(big_rhs), rhs.isneg())};
72
73      const auto new_sig {signed_lhs + signed_rhs};
74
75      return ReturnType{new_sig, lhs_exp};
76  }
```

Listing 10. Addition and Subtraction Method

3.3.3 Mul. Multiplication is the easiest of the basic operations to implement, as one might expect, because there is no requirement to handle normalization of the two numbers. To implement multiplication you need only to multiply the significands, add the exponents, compare the signs, and then construct your resulting value. The only special handling is in the multiplication of significands. We also know based on the layouts of the numbers from Listing 3 that `decimal32_t` only uses 23-bits in it's significand. This means that we only need to cast both significands to `std::uint64_t` because the result will be less than 64-bits. We let the constructor handle shrinking the value and rounding correctly as in Listing {lst:normalization. An example for 32-bit decimal types is below:

```
1  template <typename ReturnType, typename T>
2  BOOST_DECIMAL_FORCE_INLINE constexpr auto mul_impl(const T& lhs, const T&
       rhs) noexcept -> ReturnType
3  {
4      using mul_type = std::uint_fast64_t;
5      auto res_sig {(static_cast<mul_type>(lhs.full_significand()) *
       static_cast<mul_type>(rhs.full_significand()))};
6      auto res_exp {lhs.biased_exponent() + rhs.biased_exponent()};
7
8      return {res_sig, res_exp, lhs.isneg() != rhs.isneg()};
9  }
```

Listing 11. Multiplication Method

3.3.4 Div. The implementation of division is also relatively straightforward. To avoid situations in integer maths where $4/8 = 0$ we need to expand the numerator in order to ensure that we capture what will become resulting decimal digits. Similar to what we discussed above in multiplication, we cast the significand of the numerator to the next largest unsigned integer type (e.g. `std::uint32_t` becomes `std::uint64_t`), and then we multiply by the power of 10 equal to the difference be between the precision of the larger integer type, and the precision of the decimal type. We can then divide the numerator significand by the denominator significand, and subtract exponents to construct our resulting decimal number. Below is an example in code:

```
1  template <typename DecimalType, typename T>
2  BOOST_DECIMAL_FORCE_INLINE constexpr auto generic_div_impl(const T& lhs,
       const T& rhs) noexcept -> DecimalType
3  {
4      using div_type = std::uint64_t;
5
```

```
6       constexpr auto precision_offset {std::numeric_limits<div_type>::
        digits10 - precision};
7       constexpr auto ten_pow_offset {detail::pow10(static_cast<div_type>(
        precision_offset))};
8
9       const auto big_sig_lhs {lhs.sig * ten_pow_offset};
10
11      const auto res_sig {big_sig_lhs / rhs.sig};
12      const auto res_exp {(lhs.exp - precision_offset) - rhs.exp};
13
14      // Normalizes sign handling
15      bool sign {lhs.sign != rhs.sign};
16      if (BOOST_DECIMAL_UNLIKELY(res_sig == 0U))
17      {
18          sign = false;
19      }
20
21      // Let the constructor handle shrinking it back down and rounding
        correctly
22      return DecimalType{res_sig, res_exp, sign};
23  }
```

Listing 12. Division Method

4 Results

4.1 Performance

As this system is implemented in software it will never be as performant as binary floating point is on computing systems with floating-point hardware. For example the table below shows the runtime difference for comparison operations between `float`, `double`, and the types provided by this system run on an M4 Macbook Pro using homebrew Clang 20.1.8:

Type	Runtime (μs)	Ratio to `double`
`float`	64,639	1.606
`double`	40,255	1.000
`decimal32_t`	957,179	23.778
`decimal64_t`	897,409	22.293
`decimal128_t`	2,131,391	52.947
`decimal_fast32_t`	481,455	11.960
`decimal_fast64_t`	465,461	11.563

Table 2. Comparison of runtime and ratio for comparison operations

The gap in performance increases even further for the addition operation which is typically only a few cycles in hardware[1]:

What we can also see from this is that an average user should choose the fast types unless they specifically need the defined storage width. One reason that we have heard from a user in the financial sector is that they are using `decimal64_t` instead of `decimal_fast64_t` in production because it allows them to easier memcpy the value into a `std::uint64_t` and transmit.

Type	Runtime (μs)	Ratio to double
float	1,646	0.957
double	1,720	1.000
decimal32	313,219	182.104
decimal64	583,818	339.429
decimal32_fast	86,093	50.054
decimal64_fast	333,582	193.943

Table 3. Comparison of runtime and ratio for addition operations

For floating-point operations that are implemented in software like parsing and serializing numbers the performance gap is quite smaller:

Type	Runtime (μs)	Ratio to double
float	235,816	0.953
double	247,307	1.000
decimal32	366,682	1.483
decimal64	485,965	1.965

Table 4. Comparison of runtime and ratio for from_chars

Type	Runtime (μs)	Ratio to double
float	316,300	1.040
double	304,272	1.000
decimal32	406,053	1.335
decimal64	678,451	2.230

Table 5. Comparison of runtime and ratio for to_chars

5 Conclusion and Outlook

The portable C++ decimal system has been presented. The system allows for users to easily employ decimal-floating point numbers in their existing code bases. For the first time a complete and interoperable system has been fully provided.

During the course of our research and implementation we have limited ourselves to 32, 64, and 128-bit types. Further research can be conducted into making higher precision types as the properties of such numbers are specified in IEEE 754. We could also expand our range of compatibility to allow the types to be massively parallelized such as providing support for CUDA devices.

Acknowledgments

To the C++ Alliance for sponsoring the development of this library.

References

[1] ARM Limited. 2023. *ARM Architecture Reference Manual.* ARM Limited. https://developer.arm.com/documentation/ddi0487/latest ARM DDI 0487J.a (ID050623).

[2] Nelson H. F. Beebe. 2017. Historical floating-point architectures. In *The Mathematical-Function Computation Handbook - Programming Using the MathCW Portable Software Library* (1 ed.). Springer International Publishing AG, Salt Lake City, UT, USA, Chapter H, 948. https://doi.org/10.1007/978-3-319-64110-2

[3] Matt Borland, Peter Dimov, Junekey Jeon, Alexander Grund, Andrzej Krzemieński, Dmitry, Vinnie Falco, and Sam Darwin. 2024. *boostorg/charconv: Boost 1.86.0.* https://doi.org/10.5281/zenodo.13323694

[4] Michael F. Cowlishaw. 2003. Decimal Floating-Point: Algorism for Computers. In *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*. IEEE, 104–111. https://doi.org/10.1109/ARITH.2003.1207670

[5] cppreference.com contributors. 2024. Special mathematical functions. https://en.cppreference.com/w/cpp/numeric/special_functions. https://en.cppreference.com/w/cpp/numeric/special_functions Accessed: 2024-08-23.

[6] cppreference.com contributors. 2024. std::nan, std::nanf, std::nanl. https://en.cppreference.com/w/cpp/numeric/math/nan. https://en.cppreference.com/w/cpp/numeric/math/nan Accessed: 2024-08-23.

[7] John F. Hart, E. W. Cheney, Charles L. Lawson, Hans J. Maehly, Charles K. Mesztenyi, John R. Rice, Henry G. Thacher, and Christoph Witzgall. 1968. *Computer Approximations.* John Wiley & Sons, New York.

[8] IEEE. 2019. *IEEE Standard for Floating-Point Arithmetic.* Standard IEEE 754-2019. IEEE Computer Society, New York, NY, USA. https://doi.org/10.1109/IEEESTD.2019.8766229

[9] ISO/IEC. 2023. *Programming Languages – C++.* Standard ISO/IEC 14882:2023. International Organization for Standardization, Geneva, Switzerland. Sixth edition.

[10] Donald E. Knuth. 1997. *The Art of Computer Programming* (3rd ed.). Vol. 1-4B. Addison-Wesley, Reading, MA.

[11] Christopher Kormanyos. 2011. Algorithm 910: A Portable C++ Multiple-Precision System for Special-Function Calculations. *ACM Transactions on Mathematical Software (TOMS)* 37, 4, Article 45 (feb 2011), 27 pages. https://doi.org/10.1145/1916461.1916469

[12] Daniel Lemire. 2021. Number Parsing at a Gigabyte per Second. *Software: Practice and Experience* 51, 8 (2021), 1766–1785. https://doi.org/10.1002/spe.2914

[13] Jean-Michel Muller. 2016. *Elementary Functions: Algorithms and Implementation* (3 ed.). Birkhäuser, Boston. https://doi.org/10.1007/978-1-4899-7983-4

[14] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Handbook of Floating-Point Arithmetic* (2 ed.). Birkhäuser, Cham, Switzerland. https://doi.org/10.1007/978-3-319-76526-6

[15] Eric M Schwarz, John M Kapernick, and Mike F Cowlishaw. 2009. Decimal floating-point support on the IBM System z10 processor. *IBM Journal of Research and Development* 53, 1 (2009), 4:1–4:10. https://doi.org/10.1147/JRD.2009.5388557

Received XX February XXXX; revised XX March XXXX; accepted X June XXXX