Hacker's multiple-precision integer-division program in close scrutiny

Jyrki Katajainen[0000−0002−7714−5588]

[1] Department of Computer Science, University of Copenhagen, Universitetsparken 5, 2100 Copenhagen East, Denmark `jyrki@di.ku.dk`
`http://hjemmesider.diku.dk/~jyrki/`
[2] Jyrki Katajainen and Company, 3390 Hundested, Denmark

Abstract. Before the era of ubiquitous computers, the long-division method was presented in primary schools as a paper-and-pencil technique to do whole-number division. In the book "Hacker's Delight" by Warren [2nd edition, 2013], an implementation of this algorithm was given using the C programming language. In this paper we will report our experiences when converting this program to a generic program-library routine.
The highlights of the paper are as follows: (1) We describe the long-division algorithm—this is done for educational purposes. (2) We outline its implementation—the goal is to show how to use modern C++ to achieve flexibility, portability, and efficiency. (3) We analyse its computational complexity by paying attention to how the digit width affects the running time. (4) We compare the practical performance of the library routine against Warren's original. It is a pleasure to announce that the library routine is faster. (5) We release the developed routine as part of a software package that provides fixed-width integers of arbitrary length, e.g. a number of type `cphstl::ℕ<2019>` (editor's note: the non-transliterated form used in the code is `cphstl::bbbN<2019>`) has 2019 bits and it supports the same operations with the same semantics as a number of type **unsigned int**.

Keywords: Software library · Multiple-precision arithmetic · Algorithm · Long division · Description · Implementation · Meticulous analysis · Experimentation

1 Introduction

The algorithms for multiple-precision integer addition, subtraction, multiplication, and division are at the heart of algorithmics. In this paper we discuss the computer implementation of the long-division method introduced by Briggs around 1600 A.D. [`https://en.wikipedia.org/wiki/Long_division`]. The underlying ideas are even older since the Chinese, Hindu, and Arabic division methods used before that show remarkable resemblance to it [8,13].

In a positional numeral system, a string $\langle d_{\ell-1}, d_{\ell-2}, \ldots, d_0 \rangle$ of *digits* d_i, $i \in \{0, 1, \ldots, \ell-1\}$, is used to represent an integer $\boldsymbol{d}$, ℓ being the length of the representation, $d_{\ell-1}$ the most-significant digit, and d_0 the least-significant digit.

Let β, $\beta \geq 2$, denote the *base* of the numeral system. The individual digits are drawn from some bounded universe, the size of which is at least β, and the digit d_j has the *weight* β^j. In the decimal system, the digit universe is $\{0, 1, \ldots, 9\}$ and the weight of d_j is 10^j. For a general base β, the decimal *value* of $\boldsymbol{d}$ is $\sum_{j=0}^{\ell-1} d_j \cdot \beta^j$. As is customary, in this representation the leading zero digits (0) may be omitted, except when representing number zero ($\mathbf{0}$).

In the computer representation of a number, the digit width is often selected to be in harmony with the word size of the underlying hardware. We use `W` to denote the type of the digits and we assume that the width of `W` is a power of two. The numbers themselves are arrays of digits of type `W`. The length of these arrays can be specified at compile time (`std::array` in C++), or the length can be varying and may change at run time (`std::vector` in C++). The memory for these arrays can be allocated from the stack at run time (so-called C arrays) or from the heap relying on the memory-allocation and memory-deallocation methods provided by the operating system. Since memory management is not highly relevant for us, we will not discuss this issue here.

In the division problem for whole numbers (non-negative integers), the task is to find out how many times a number $\boldsymbol{y}$ (*divisor*) is contained in another number $\boldsymbol{x}$ (*dividend*). Throughout the paper, we use the division operator / to denote the whole-number division and we assume that $\boldsymbol{y} \neq \mathbf{0}$ since division by $\mathbf{0}$ has no meaning. That is, the output of $\lfloor \boldsymbol{x}/\boldsymbol{y} \rfloor$ is the largest whole number $\boldsymbol{q}$ (*quotient*) for which the inequality $\boldsymbol{q} * \boldsymbol{y} \leq \boldsymbol{x}$ holds. Throughout the paper, we use n to denote the *length* of the dividend (the number of its digits) and m the length of the divisor. After computing the quotient, the *remainder* $\boldsymbol{x} - \boldsymbol{q} * \boldsymbol{y}$ can be obtained by a single long multiplication and long subtraction. We ignore the computation of the remainder, but we acknowledge that a routine `divmod` that computes both the quotient and the remainder at the same time could be handy.

The main motivation for this study was the desire to implement a program package for the manipulation of multiple-precision integers. In our application (see [3]), we only needed addition, subtraction, and multiplication for numbers whose length was two or three words. When making the package complete and finishing the job, the implementation of the division algorithm turned out to be a non-trivial task. We are not the first to make this observation (see, e.g. [1]).

First, we reviewed the presentation of the long-division algorithm in "The Art of Computer Programming" (Volume 2) [6, Section 4.3.1]. Knuth described the algorithm (Algorithm D), proved its correctness (Theorem B), analysed its complexity, and gave an implementation (Program D) using his mythical MIX assembly language. The paper by Pope and Stein [11] was one of the significant sources used by him. Under reasonable assumptions, Knuth estimated that, in the average case, the program will execute about $30\, n \cdot m + O(n + m)$ MIX instructions. (Before reading Knuth's book, check the official errata available at [`https://www-cs-faculty.stanford.edu/~knuth/taocp.html`]—this can save you some troubles later.)

Next, we looked at the Pascal implementation described by Brinch Hansen [1] and the C implementation described by Warren in the book "Hacker's Delight"

[12] (errata can be found at [`https://www.hackersdelight.org/`]). In particular, Warren carefully examined many implementation details so we decided to base our library implementation on his programs (the source code is available at [`https://www.hackersdelight.org/`]). We looked at other sources as well, but very quickly we got back to Algorithm D or some of its variants. When the numbers are not longer than a few thousand digits, the long-division algorithm should be good enough for most practical purposes. Although its asymptotic complexity is high $O(n \cdot m)$, the leading constant in the order notation is small.

In this write-up, we report our observations when implementing the division routine for the multiple-precision integers provided by the CPH STL [`http://www.cphstl.dk/`]. We put emphasis on the following issues:

Portability. In the old sources the digit universe is often fixed to be small. For example, in Warren's implementation the width of digits was set to 16 bits. For our implementation the digit width can be any power of two—it should just be specified at compile time. We wrote the programs using C++. This made it possible to hide some of the messy details inside some few subroutines called by the high-level code.

Analysis. Instead of the MIX cost used by Knuth or the RISC cost used by Warren, we analyse the Intel cost—the number of Intel assembler instructions—of the long-division routine as a function of the number of the bits in the inputs and the word size of the underlying computer. (Before reading any further, you should stop for a moment to think about what would be a good data type `W` for the digits when dividing an N-bit number with an $\frac{N}{2}$-bit number.)

Efficiency. We perform some experiments to check the validity of our back-of-the-envelope calculations in a real machine. The tests show unanimously that our program—with larger digit widths—is faster than Warren's program. And because of adaptability, it should be relatively easy to modify the code—if at all necessary—if the underlying hardware changes.

2 Long-Division Algorithm

Let $\odot$ be one of the operations supported by the C++ programming language for integers, e.g. $==$, $<$, $+$, $-$, $*$, $/$, $\%$, $>>$, $<<$, $\sim$ (**compl**), **&** (**bitand**), or $||$ (**bitor**). To understand the beauty of the division algorithm, we use the notation $\boldsymbol{\odot(n, m)}$ to denote a subroutine that performs the $\odot$ operation when the first operand is an n-digit number and the second operand (if any) an m-digit number.

2.1 Software Stack

To perform the operation $\boldsymbol{/(n, m)}$, the long-division algorithm needs the following subroutines:

$\boldsymbol{\odot(1)}$**,** $\odot \in \{\sim, \mathbf{nlz}\}$**.** The primitive $\sim$ computes the bitwise complement of a digit and **nlz** the number of leading 0 bits in a digit. We assume that these primitives are available in hardware or provided by the environment.

$\boldsymbol{\odot(1,1)}$, $\boldsymbol{\odot} \in \{==, <, /, \%, >>, <<, \&, ||\}$. We assume that these operations are also available in hardware or provided by the environment.

$\boldsymbol{+(1,1)}$. We assume that this operation is a built-in primitive. The overflow bit (*carry*) can be computed by checking whether the sum is smaller than one of the operands ($\boldsymbol{<(1,1)}$ operation) [12, Section 2-16].

$\boldsymbol{-(1,1)}$. We assume that this operation is a built-in primitive. The underflow bit (*borrow*) can be computed by checking whether the first operand is smaller than the second ($\boldsymbol{<(1,1)}$ operation) [12, Section 2-16].

$\boldsymbol{*(1,1)}$. We assume that this operation is a built-in primitive, but the output consists of two digits so the higher-order digit must be computed separately. A routine that computes the higher-order digit without overflows is described in [12, Fig. 8-2]. (It requires 16 RISC instructions.)

$\boldsymbol{+(2,1)}$. This operation involves two $\boldsymbol{+(1,1)}$ operations and one $\boldsymbol{<(1,1)}$ operation to forward the carry bit (if any) from the first position to the second. The operation is always used in a context where the overflow can be ignored.

$\boldsymbol{/(2,1)}$. The operation is only needed in a context where the output is one digit long. In principle, this operation implements the division tables which are the reversal of the multiplication tables we learnt at school. This operation is the most complicated subroutine; an implementation is given in [12, Fig. 9-4]. (According to Warren's analysis, for uniformly distributed random numbers, this operation executes about 52 RISC instructions.)

$\boldsymbol{*(n,1)}$. This operation can be accomplished in a single scan over the first operand by invoking n times the $\boldsymbol{*(1,1)}$ operation, forwarding the higher-order digit from the previous position and adding it to the result of the multiplication with a $\boldsymbol{+(2,1)}$ operation [1, Algorithm 2]. This form of multiplication is always used in a context where the overflow can be ignored.

$\boldsymbol{<(n,n)}$. This operation is a simple scan over the digits starting from the most-significant end [1, Algorithm 6]. The first position where the digits differ is found (if any) using the $\boldsymbol{==(1,1)}$ operation and at the found position the $\boldsymbol{<(1,1)}$ operation is applied to get the answer.

$\boldsymbol{-(n,n)}$. This operation can be accomplished in one scan by performing n $\boldsymbol{+(1,1)}$ operations, n $\boldsymbol{-(1,1)}$ operations, and $2n$ $\boldsymbol{<(1,1)}$ operations to handle the borrow from the previous position [1, Algorithm 7]. This form of subtraction is always used in a context where the underflow can be ignored.

Since the computational complexity of the long-division algorithm will be determined by the routines $\boldsymbol{*(n,1)}$, $\boldsymbol{<(n,n)}$, and $\boldsymbol{-(n,n)}$, we give them their own names `product`, `is_less`, and `difference`, respectively.

2.2 Algorithm Description

Let us consider how the division problem can be solved when the dividend is $\boldsymbol{x} = \langle x_{n-1}, x_{n-2}, \ldots, x_0\rangle$ and the divisor $\boldsymbol{y} = \langle y_{m-1}, y_{m-2}, \ldots, y_0\rangle$. We assume that n and m are the real lengths of the numbers so that $x_{n-1} \neq 0$ and $y_{m-1} \neq 0$. Recall that the digits are of type `W` and let w be the width of `W` in bits.

At a high level, the long-division algorithm is simple: it computes the quotient digits one at the time starting from the most-significant end. The basic complication is the need of a good estimate $\widehat{\mathrm{q}}$ for the next quotient digit. When this is available, the partial remainder can be updated and the computation can proceed to the next digit.

To get a reasonable estimate for the next quotient digit, the key algorithmic idea is *normalization* [11]: this means that the divisor is cast into the form where its most significant digit is higher than or equal to 2^{w-1}. One way to achieve this is to multiply both the dividend and the divisor with some factor f, which makes the most-significant digit of the divisor large enough. Let $\overline{\boldsymbol{x}} = f * \boldsymbol{x}$ and $\overline{\boldsymbol{y}} = f * \boldsymbol{y}$. Since $\lfloor \overline{\boldsymbol{x}} / \overline{\boldsymbol{y}} \rfloor = \lfloor \boldsymbol{x} / \boldsymbol{y} \rfloor$, the quotient for the normalized numbers is the same as that for the original numbers. Knuth used the factor $f = \lfloor 2^w / (y_{m-1} + 1) \rfloor$ (see the errata of [6, Section 4.3.1]). Warren [12, Fig. 9-4] used the factor $f = 2^\sigma$, where σ is the number of leading 0 bits in y_{m-1}.

During the execution of the algorithm, the *partial remainder* is maintained in $\boldsymbol{u} = \langle u_n, u_{n-1}, \ldots, u_0 \rangle$ which is initialized to contain the normalized dividend $f * \boldsymbol{x}$. The normalized divisor is maintained in $\boldsymbol{v} = \langle v_m, v_{m-1}, \ldots, v_0 \rangle$. In the main loop of the algorithm, the loop index j goes down from $n - m$ to 0. We call the subrange of length $m+1$ $\langle u_{j+m}, u_{j+m-1}, \ldots, u_j \rangle$ the *active part* of the partial remainder. Then the operation $\boldsymbol{/(2,1)}$ with the first two digits $\langle u_{j+m}, u_{j+m-1} \rangle$ of the active part and v_{m-1} is used to compute an estimate $\widehat{\mathrm{q}}$ for the next quotient digit. This estimate is the correct quotient digit, or it is one or two too high [6, Theorem B]. Collins and Musser [2] proved that for random numbers, with high probability, the estimate is correct or off by one.

These results have been improved in several ways: (1) Mifsud [9] (see an addendum in [10]) proved that with more aggressive normalization the estimate can be guaranteed to be correct or off by one. (2) Krishnamurthy and Nandi [7] obtained the same result by using the prefixes of 3 and 2 digits when calculating the estimate. So both of these approaches guarantee that not more than one correction is required to obtain the true quotient digit. (3) Also, people have tried to find conditions under which the normalization can be skipped (see, for example, [7,9]). Even if the use of $\boldsymbol{v}$ can be avoided, temporary storage is still needed to store the active part of the partial remainder $\boldsymbol{u}$ and the product $\boldsymbol{p}$ of $\widehat{\mathrm{q}}$ and the (normalized) divisor.

Now we can describe the algorithm in detail:

(1) If $\boldsymbol{x} < \boldsymbol{y}$, return $\boldsymbol{0}$ as the answer. This comparison is a generalization of the $\boldsymbol{<(n,n)}$ operation where the operands are not necessarily of the same length. It involves a synchronous scan over the digits starting from the end of the longer string. After this step we can be sure that $n \geq m$.
(2) Allocate space for the quotient $\boldsymbol{q} = \langle q_{n-m}, q_{n-m-1}, \ldots, q_0 \rangle$ and fill it with zeros.
(3) Allocate space for the partial remainder $\boldsymbol{u} = \langle u_n, u_{n-1}, \ldots, u_0 \rangle$ and copy $\boldsymbol{x}$ there; observe that $\boldsymbol{u}$ is one longer, so u_n is set to zero.
(4) Allocate space for the normalized divisor $\boldsymbol{v} = \langle v_m, v_{m-1}, \ldots, v_0 \rangle$ and copy $\boldsymbol{y}$ there; v_m is needed to make this string $m + 1$ long, so v_m is set to zero.
(5) Compute the number of leading 0 bits in the digit y_{m-1}. Let this be σ.

(6) Shift the bits of $\boldsymbol{u}$ σ positions to the left. This operation is a special case of the $\boldsymbol{*(n+1,1)}$ operation where the multiplier is 2^{σ}. Since $\boldsymbol{u}$ is one digit longer than $\boldsymbol{x}$ and $\sigma < w$, no overflow is possible.
(7) Shift the bits of $\boldsymbol{v}$ σ positions to the left. Naturally, no overflow is possible and after this operation the leading bit of v_{m-1} is set as required.
(8) Compute now the digits of $\boldsymbol{q}$, one by one, by letting the loop index j go down from $n-m$ to 0.
   (a) Calculate an estimate $\widehat{\mathrm{q}}$ for the quotient digit by invoking the $\boldsymbol{/(2,1)}$ operation with the arguments $\langle u_{j+m}, u_{j+m-1}\rangle$ and v_{m-1}. However, if $u_{j+m} \geq v_{m-1}$, set $\widehat{\mathrm{q}}$ equal to $2^{w}-1$ without performing the division.
   (b) Compute the product of $\boldsymbol{v}$ and $\widehat{\mathrm{q}}$ by invoking the $\boldsymbol{*(m+1,1)}$ operation. Keep the result temporarily in $\boldsymbol{p} = \langle p_m, p_{m-1}, \ldots, p_0\rangle$.
   (c) Check if the estimate is too large by invoking the $\boldsymbol{<(m+1,m+1)}$ operation for the active part of the partial remainder $\langle u_{j+m}, u_{j+m-1}, \ldots, u_j\rangle$ and $\boldsymbol{p}$.
   (d) If the estimate was too large, make it one smaller, subtract $\boldsymbol{v}$ from $\boldsymbol{p}$ by performing the $\boldsymbol{-(m+1,m+1)}$ operation, and go back to Step 8c.
   (e) Otherwise, set q_j equal to $\widehat{\mathrm{q}}$. Furthermore, update the partial remainder by subtracting the computed product $\boldsymbol{p}$ from the active part by invoking the $\boldsymbol{-(m+1,m+1)}$ operation. Hereafter we can proceed to the computation of the next quotient digit.
(9) Release the space allocated for $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{p}$.
(10) Return $\boldsymbol{q}$ as the result of the computation.

2.3 Asymptotic Analysis

In this algorithm, Steps (1)–(7) all involve sequential scans over the digit strings. If the digits can be processed at unit cost, the amount of work done is $O(n+m)$. Most of the work is done in Step 8. Of the substeps, Step 8b calls the function `product`, Step 8c the function `is_less`, and Steps 8d and 8e the function `difference`. The arguments are of length $m+1$. Each of these operations involves a linear scan over the digits. Therefore, the asymptotic complexity of the algorithm is $O((n-m)\cdot m + n + m)$.

3 Implementation

In this section we describe our implementation of the long-division algorithm. The source code is extracted from the CPH STL so, unfortunately, it contains some noise that has to be explained first.

Standard library. A good documentation of the facilities available at the C++ standard library can be found at [`https://en.cppreference.com/`].

Constraints and concepts. In the code some requirements are specified for the template arguments to ensure that the components are used in a correct way. Here we rely on the features drafted in the upcoming C++2a standard, but some compilers support them already now.

Type functions. In the code some metaprogramming tools are used; these are taken from the CPH MPL (Copenhagen metaprogramming library) [4]. A *type function* maps a type to some value or to some type, and this computation is done at compile time. By convention, a type function, the name of which begins with `is_`, returns a Boolean value. As concrete examples, consider the following type functions specified for some type `W`:

(1) The built-in function **sizeof**(`W`) gives the size of the objects of type `W`, measured in bytes. Unfortunately, for this type function the syntax is not the same as that preferred in the CPH MPL.

(2) The type function `cphmpl::width<W>` returns the width of the objects of type `W`, measured in bits. In our test computer, the compiler will replace all occurrences of `cphmpl::width<int>` in the code with the number 32.

(3) The type function `cphmpl::twice_wider<W>` specifies an alias for the type, the width of which is twice as large as that of `W`. For example, `cphmpl::twice_wider<cphstl::ℕ<512>>` is an alias for `cphstl::ℕ<1024>`.

Ranges. The digit strings given for the programs can be stored in a `std::array`, in a `std::vector`, in a C array, or in any other container—or part of it—that supports (bidirectional) iterators. A *range* specifies such a sequence. To manipulate the digits, it must be possible to use a range as an argument for the functions `std::begin`, `std::cbegin`, `std::end`, `std::cend`, `std::size`, and `std::empty`. With this abstraction, the programs are independent of the representation of the digit strings.

Hidden details. The code for some functions is omitted on purpose. Many of the omitted functions defined inside the **namespace** `cphstl::detail` work for an arbitrary numeric type, but they are overloaded to work more efficiently for the standard integer types.

3.1 Function is_less

The implementation of function `is_less` is given in Listing 1. Starting from the most significant digit, the purpose is to find the first position where the two strings differ and then use the found digits to determine the answer. The critical inner loop is in lines 15–18.

Listing 1. Function is_less in C++.

```
1  template<typename L, typename R>
2  requires
3  /* 1 */ cphmpl::specifies_range<L> and
4  /* 2 */ cphmpl::specifies_range<R> and
5  /* 3 */ std::is_same_v<cphmpl::value<L>, cphmpl::value<R>>
6  bool is_less(L const& lhs, R const& rhs) {
7    // check whether lhs < rhs or not
8    assert(std::size(lhs) == std::size(rhs));
9    assert(not std::empty(lhs));
10   using I = cphmpl::const_iterator<L>;
11   using J = cphmpl::const_iterator<R>;
12   I p = std::cend(lhs);
13   J q = std::cend(rhs);
```

```
  I first = std::cbegin(lhs);
  do {
    --p;
    --q;
  } while (p != first and *p == *q);
  return *p < *q;
}
```

We declared the digits to be of type **unsigned long long int**, the size of which was 8 bytes, and asked the compiler to generate the assembler code for the inner loop of `is_less`. The inner loop had 7 instructions. A micro-benchmark that was used to verify this count compared two equal numbers. The test revealed that, when the digits were of type **unsigned char**, the compiler could optimize the code so that the execution only required 0.17 instructions per digit. This optimization was not done for the other standard types.

3.2 Function `difference`

In long division, it is only necessary to do the subtraction $\boldsymbol{x} - \boldsymbol{y}$ when the two numbers have the same length and when $\boldsymbol{x} \geq \boldsymbol{y}$. Also, it is not necessary to keep the old value. Therefore, we implemented the operation $\boldsymbol{x}$ −= $\boldsymbol{y}$ in addition to the general subtraction. The C++ code for this is given in Listing 2. Here the inner loop is in lines 18–26.

Listing 2. Function `difference` in C++.

```
1  template<typename L, typename R>
2  requires
3  /* 1 */ cphmpl::specifies_range<L> and
4  /* 2 */ cphmpl::specifies_range<R> and
5  /* 3 */ std::is_same_v<cphmpl::value<L>, cphmpl::value<R>> and
6  /* 4 */ cphmpl::is_unsigned<cphmpl::value<L>>
7  void difference(L& minuend, R const& subtrahend) {
8    // compute minuend -= subtrahend
9    assert(std::size(minuend) == std::size(subtrahend));
10   assert(not std::empty(minuend));
11   using I = cphmpl::iterator<L>;
12   using J = cphmpl::const_iterator<R>;
13   using W = cphmpl::value<L>;
14   I p = std::begin(minuend);
15   J q = std::cbegin(subtrahend);
16   I past = std::end(minuend);
17   bool borrow = 0;
18   while (p != past) {
19     W t = *q + W(borrow);
20     bool overflow = (t < *q);
21     bool underflow = (*p < t);
22     *p = *p - t;
23     borrow = overflow or underflow;
24     ++p;
```

```
    ++q;
  }
}
```

Again we let the compiler generate the assembly-language translation when the digits were of type **unsigned long long int**. The inner loop had 15 instructions. When the digits were of type **unsigned char**, the compiler could optimize the code so that the execution only required about 12 instructions per digit.

3.3 Function `product`

In long division, only a restricted form of multiplication is needed where a number $\boldsymbol{x}$ is multiplied by a single digit. Furthermore, it is not allowed to modify $\boldsymbol{x}$ so the result must be saved somewhere else. We assume that the caller has allocated space for the result. Listing 3 gives the C++ code that does this multiplication.

Listing 3. Function `product` in C++.

```
1  template<typename L, typename R, typename W>
2  requires
3  /* 1 */ cphmpl::specifies_range<L> and
4  /* 2 */ cphmpl::specifies_range<R> and
5  /* 3 */ cphmpl::is_unsigned<W> and
6  /* 4 */ std::is_same_v<cphmpl::value<L>, W> and
7  /* 5 */ std::is_same_v<cphmpl::value<R>, W>
8  void product(L& result, R const& multiplicand, W const& factor) {
9    // compute result = multiplicand * factor
10   assert(std::size(result) == std::size(multiplicand));
11   using D = cphmpl::twice_wider<W>;
12   using I = cphmpl::iterator<L>;
13   using J = cphmpl::const_iterator<R>;
14   J first = std::cbegin(multiplicand);
15   J past = std::cend(multiplicand);
16   W carry = W();
17   I q = std::begin(result);
18   for (J p = first; p != past; ++p, ++q) {
19     D t = cphstl::detail::multiply<D>(*p, factor);
20     t = cphstl::detail::add(t, carry);
21     *q = cphstl::detail::lower_half<W>(t);
22     carry = cphstl::detail::upper_half<W>(t);
23   }
24 }
```

Here `W` is the type of the digits and `D` is an alias for a type that is twice as wide as `W`. The function `cphstl::detail::multiply` performs the operation $\boldsymbol{*(1,1)}$ and the function `cphstl::detail::add` the operation $\boldsymbol{+(2,1)}$. Finally, the remaining functions `cphstl::detail::lower_half` and `cphstl::detail::upper_half` are used to get from a digit of type `D` its two halves of type `W`.

The inner loop is in lines 18–23. When **sizeof**(`W`) was 8 and `D` was an alias of **unsigned** `__int128`—an extension supported by the `g++` compiler, the assembly-language translation of this loop contained 10 instructions. On the other hand, when `D` was an alias of `std::array<W, 2>`, the inner loop contained 26 instructions. For **unsigned char** the instruction count was 9, and for **unsigned short** and **unsigned int** it was 10. For the digit widths 128 and 256, the instruction count dropped to around 6 which could be explained by the fact that the compiler had turned on the streaming SIMD extensions (SSE), allowing parallel operations on four values per instruction.

3.4 Main Loop

After these initial exercises, we can peek inside the long-division program. Its main loop is shown in Listing 4. The meaning of most functions should be clear by their names. The function `cphstl::detail::halves_together` concatenates two digits and the function `cphstl::detail::divide` performs the $\mathbf{/(2,1)}$ operation. Because of the **if** test before this division operation, the output is always a single digit and the upper half can be discarded.

Listing 4. The main loop of the long-division program in C++; array `u` contains the partial remainder, array `v` the normalized divisor, and array `p` is for temporary use.

```
auto normalized_divisor = cphstl::range(&v[0], &v[m + 1]);
auto temporary = cphstl::range(&p[0], &p[m + 1]);
auto q = std::begin(quotient);
std::advance(q, n - m);

for (int j = n - m; j ≥ 0; --j, --q) {
  auto active_part = cphstl::range(&u[j], &u[j+m+1]);
  W q̂ = compl W(); // estimate for the quotient digit
  if (u[j+m] < v[m-1]) {
    D t = cphstl::detail::halves_together<D>(u[j+m-1], u[j+m]);
    t = cphstl::detail::divide(t, v[m-1]);
    q̂ = cphstl::detail::lower_half<W>(t);
  }
  cphstl::detail::product(temporary, normalized_divisor, q̂);
  while (cphstl::detail::is_less(active_part, temporary)) {
    --q̂; // correction; estimate may be 1 or 2 too large
    cphstl::detail::difference(temporary, normalized_divisor);
  }
  *q = q̂;
  cphstl::detail::difference(active_part, temporary);
}
```

4 Meticulous Analysis

After describing the long-division program, we can analyse its performance. All the processing is sequential, so we are mainly interested in the number of instructions executed. In the analysis we keep the number of digits (n) fixed, but vary the width of the digits.

Table 1. Summary of the instruction counts (per digit) determined experimentally for the performance-critical functions.

Digit width	`is_less`	`difference`	`product`
8	0.17	12.30	9.17
16	7.16	14.28	10.16
32	7.16	14.24	10.15
64	7.18	15.24	26.17
128	10.40	33.51	6.39
256	16.68	71.88	6.67
512	29.26	148.62	2 874
1024	54.37	317.04	14 339

In Table 1, we summarize the instruction counts that were measured for the efficiency-determining functions. In the reported counts, the total number of instructions executed is divided by n. In the micro-benchmarks, (1) `is_less` compared two equal numbers; (2) `difference` processed two random numbers, except that the first was made larger by resetting the most significant digits; and (3) `product` multiplied a long random number with a random digit. These counts are approximations, but they are firmly linked to the generated assembler code.

Assume now that N is a power of two and that we want to divide an N-bit number with an $\frac{N}{2}$-bit number. When the word size is α, our theoretical analysis shows that the running time of the long-division program should be proportional to $\frac{N}{2\alpha} \cdot \frac{N}{2\alpha}$. This analysis is based on the assumption that digits can be processed at unit cost. The micro-benchmarks show that—in the test environment—this assumption is valid up to 64, or maybe all the way to 256.

Assuming that we rely on the more aggressive normalization proposed by Mifsud [9], the estimate is correct or off by one. Then, in the worst-case scenario, in each iteration of the main loop the functions `is_less` and `product` are called once, and `difference` is called twice. Thus, for the word size α ($\alpha \leq 64$), N-bit dividend, and $\frac{N}{2}$-bit divisor, the worst-case Intel cost of the long-division program is $\frac{1}{4} \cdot (7.18 + 2 \cdot 15.24 + 26.17) \cdot \left(\frac{N}{\alpha}\right)^2 + O\left(\frac{N}{\alpha}\right)$, which is $15.95 \cdot \left(\frac{N}{\alpha}\right)^2 + O\left(\frac{N}{\alpha}\right)$.

5 Integration with the library

The class templates `cphstl`::$\mathbb{N}$ and `cphstl`::$\mathbb{Z}$ are designed to provide fixed-width integers of arbitrary length [5]. The number of bits (`b`) used in the representation is specified at compile time. Let us use `U` as a shorthand for the standard type **unsigned long long int** and let $\boldsymbol{\alpha}$ = `cphmpl::width<U>`. The class template `cphstl`::$\mathbb{N}$ is written in two parts using constraint-based overloading.

(1) When $0 < \mathtt{b} \leq \boldsymbol{\alpha}$, the classes `cphstl`::$\mathbb{N}$`<b>` are just thin wrappers around the standard unsigned integer types (Listing 5). If `b` is not a power of two, additional sanitation is needed to perform the calculations modulo $2^{\mathtt{b}}$.

Listing 5. An extract from the **private** part of `cphstl::`$\mathbb{N}$`<b>` for $0 < \mathtt{b} \leq \boldsymbol{\alpha}$.

```
using uints = cphmpl::typelist<unsigned char, unsigned short int,
    ↪ unsigned int, unsigned long int, unsigned long long int>;
using W = uints::get<detail::first_wide_enough<uints, b>()>;

W data;
```

(2) When $\mathtt{b} > \boldsymbol{\alpha}$, an integer is represented as a `std::array<U, n>`, where $\mathtt{n} = \lfloor(\mathtt{b} + \boldsymbol{\alpha} - 1)/\boldsymbol{\alpha}\rfloor$ (Listing 6). The long-division algorithm is in action first when the numbers are wider than $\boldsymbol{\alpha}$.

Listing 6. An extract from the **private** part of `cphstl::`$\mathbb{N}$`<b>` for $\mathtt{b} > \boldsymbol{\alpha}$.

```
using U = unsigned long long int;

static constexpr std::size_t α = cphmpl::width<U>;
static constexpr std::size_t n = (b + α − 1) / α;

std::array<U, n> data;
```

In the first place, we needed the long-division program for the implementation of **operator**/ for the class templates `cphstl::`$\mathbb{N}$ and `cphstl::`$\mathbb{Z}$. Soon this program became an important test-bed for the whole library since the functions inside the library should work for these fixed-width integers themselves. In particular, in long division, it is now possible to choose the digits to be of type `cphstl::`$\mathbb{N}$`<b>` for arbitrary positive integer `b` that is a power of two.

To get some insight into the program transformations involved, when converting Warren's implementation [12, Chapter 9] into a generic library routine, look at the following code extracts taken from Hacker's Delight (Listing 7) and the CPH STL (Listing 8), respectively. When `W` is an alias of **unsigned int**, the assembler code generated by the compiler should be identical for both, but the latter works for any unsigned integer type and it can even be faster.

Listing 7. An extract from the function `divlu` in [12, Fig. 9-3].

```
// v is the divisor of type unsigned int

unsigned vn0, vn1;
int s;

s = nlz(v);              // 0 ≤ s ≤ 31
v = v << s;              // Normalize divisor.
vn1 = v >> 16;           // Break divisor into
vn0 = v & 0xFFFF;        // two 16−bit digits.
```

Listing 8. An extract from the function `divide_long_unsigned` in the CPH STL.

```
// W is a template parameter
// v is the divisor of type W

```

```
constexpr std::size_t w = cphmpl::width<W>;
constexpr W oooffff = cphstl::some_trailing_ones<w / 2, W>;

std::size_t const s = cphstl::leading_zeros(v);
v = v << s;
W const vn1 = v >> (w / 2);
W const vn0 = v bitand oooffff;
```

The functions `nlz` [12, Section 5-3] and `cphstl::leading_zeros` compute the number of leading 0 bits in the representation of a digit. In the CPH STL, this function is overloaded to work differently depending on the type of the argument. For the standard integer types, it can even call an intrinsic function that will be translated into a single hardware instruction[3]. There is also a **constexpr** form that computes the value at compile time if the argument is known at that time.

6 Benchmarking

In the following we will explain in more detail how we evaluated the quality of the division routines in the CPH STL.

6.1 Computing Environment

All the experiments were done on a personal computer that run Linux. The programs were written in C++ and the code was compiled using the `g++` compiler. The hardware and software specifications of the system were as follows.

Processor. Intel® Core™ i7-6600U CPU @ 2.6 GHz × 4
Word size. 64 bits
Operating system. Ubuntu 18.04.1 LTS
Linux kernel. 4.15.0-43-generic
Compiler. `g++` version 8.2.0—GNU project C++ compiler
Compiler options. `-O3 -Wall -Wextra -std=c++2a -fconcepts -DNDEBUG`
Profiler. `perf stat`—Performance analysis tool for Linux
Profiler options. `-e instructions`

6.2 Small Numbers

In our first experiment, we wanted to test how well the operations for the types `cphstl::`$\mathbb{N}$`<b>` perform. The benchmark was simple: For an array `x` of n digits and a digit `f`, execute the assignment `x[i] = x[i]` $\odot$ `f` for all `i` $\in \{0, 1, \ldots, n-1\}$. In the benchmark the number of instructions executed, divided by n, was measured for different digit types and operators $\odot \in \{+, -, *, /\}$.

The obtained instruction counts are reported in Table 2. Here the absolute values are not important due to loop overhead; one should look at the relative values instead. From these results, we make two conclusions:

[3] The Windows support of the bit tricks was programmed by Asger Bruun.

Table 2. The number of instructions executed (per operation) on an average when performing scalar-vector arithmetic for different types; in the implementation of `cphstl`::ℕ<128> the GNU extension **unsigned** `__int128` was not used.

Type	+	−	∗	/
unsigned char	0.40	0.40	0.90	6.02
unsigned short int	0.46	0.46	0.46	4.53
unsigned int	0.89	0.89	2.39	4.52
unsigned long long int	1.77	1.77	5.77	4.53
unsigned `__int128`	5.53	5.53	7.04	19.55
`cphstl`::ℕ<8>	0.25	0.25	0.72	6.02
`cphstl`::ℕ<16>	0.46	0.46	0.46	7.02
`cphstl`::ℕ<24>	1.14	1.14	2.77	7.02
`cphstl`::ℕ<32>	0.89	0.89	2.39	7.02
`cphstl`::ℕ<48>	2.27	2.27	6.27	7.03
`cphstl`::ℕ<64>	1.77	1.77	5.77	7.03
`cphstl`::ℕ<128>	5.54	7.54	24.07	18.12
`cphstl`::ℕ<256>	18.06	27.06	161.9	49.37
`cphstl`::ℕ<512>	38.11	81.11	407.6	73.61
`cphstl`::ℕ<1024>	96.21	179.2	1396	129.3

(1) The `g++` compiler works well! When wrapping the standard types into a class, the abstraction penalty is surprisingly small. Division is somewhat slower due to the check if the divisor is zero, which is done to avoid undefined behaviour.
(2) On purpose, in the implementation of the type `cphstl`::ℕ<128>, we did not rely on the extension **unsigned** `__int128`. Instead the double-length arithmetic was implemented as explained in [12, Section 2-16, Section 8-2, and Fig. 9.3]. In particular, multiplication is slow compared to **unsigned** `__int128`.

6.3 Large Numbers

In our second experiment, we considered the special case where the dividend was an N-bit number and the divisor an $\frac{N}{2}$-bit number, and we run the long-division programs for different values of N and digit widths. In the benchmark, we generated two random numbers and measured the number of instructions executed. When reporting the results, we scaled the instruction counts using the scaling factor $\left(\frac{N}{64}\right)^2$. Here the rationale is that any program should be able to utilize the power of the words native in the underlying hardware. The test computer was a 64-bit machine. We fixed six measurement points: $N \in \left\{2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}\right\}$. The obtained results are reported in Table 3.

We expected to get the best performance when the width of the digits matches the word size, but wider digits produced better results. As the instruction counts for Warren's program [12, Fig. 9-3] indicate, the choice 16 for the digit width is based on old technological assumptions. The figures for the width 16 also reveal that we have not followed the sources faithfully. (We have not used Knuth's optimization in Step D3 of Algorithm D [6, Section 4.3.1] and we have

Table 3. The performance of the long-division programs for different digit widths, measured in the number of instructions executed when processing two random numbers of N and $\frac{N}{2}$ bits. The values indicate the coefficient C in the formula $C \cdot \left(\frac{N}{64}\right)^2$.

Digit width	$N = 2^{12}$	$N = 2^{14}$	$N = 2^{16}$	$N = 2^{18}$	$N = 2^{20}$	$N = 2^{22}$
8	447.1	390.5	427.2	361.9	410.5	392.7
16	110.9	97.1	124.1	113.9	104.6	135.0
32	34.2	32.1	28.6	30.8	26.6	29.1
64	14.4	12.5	10.9	12.6	11.9	11.3
128	5.8	3.2	2.6	2.4	2.4	2.4
256	13.5	4.3	1.9	1.3	1.2	1.2
512	15.5	3.8	1.3	0.7	0.6	0.6
1024	36.1	18.5	14.7	13.9	13.7	13.6
16 [12, Fig. 9-3]	63.3	60.8	60.2	60.0	60.0	60.0

not fusioned the loops in `product` and `difference`.) The slowdown may also be due to abstraction overhead. Nonetheless, for larger digit widths, in these tests the library routine performed significantly better than Warren's program.

7 Final Remarks

The long-division program is based on the concept of digits. In a generic implementation, the type of digits is given as a template parameter so it will be fixed at compile time. If the lengths of the inputs are known beforehand, the code can be optimized to use the best possible digit width `b`. According to our experiments, for large values of N, the best performance is obtained for large values of `b`. The optimum depends on the operations supported by the hardware.

Seeing the program hierarchically, several levels of abstractions are visible:

User level. operator/ provided by the types `cphstl`::ℕ<`b`> and `cphstl`::ℤ<`b`> for any specific width `b`.

Implementation level. Operation $\boldsymbol{/(n, m)}$ where n and m are the number of digits in the operands.

Efficiency-determining functions. Operations $\boldsymbol{<(n, n)}$ (`is_less`), $\boldsymbol{-(n, n)}$ (`difference`), and $\boldsymbol{*(n, 1)}$ (`product`).

Intermediate level. Operation $\boldsymbol{+(2, 1)}$, which will not overflow, and operation $\boldsymbol{/(2, 1)}$, which just needs to work when the output is a single digit.

Overflowing primitives. Operations $\boldsymbol{+(1, 1)}$ and $\boldsymbol{-(1, 1)}$ can overflow or underflow by one bit, and operation $\boldsymbol{*(1, 1)}$ has a two-digit output.

Safe primitives. Operations $\boldsymbol{\odot(1, 1)}$, $\odot \in \{==, <, /, \%, >>, <<, \&, ||\}$, must also be provided, but they cannot overflow.

Bit-manipulation primitives. Unary operations $\boldsymbol{\odot(1)}$, $\odot \in \{\sim, \mathbf{nlz}\}$ are needed for division, but the library supports other bit tricks as well.

For many library functions, there exist several overloaded versions to get the best match with the instructions provided by the underlying hardware. Here the keywords are constraint-based function overloading and template specialization.

When dividing an N-bit number by an $\frac{N}{2}$-bit number, we determined a good digit width `b` experimentally. Instead of using this idea only once, one could use a divide-and-conquer approach where the digits are subdivided into subdigits and the method is applied recursively. As the results of our experiments suggest, for large values of N, this approach may have practical value. At least it would take the hacking to another level.

References

1. Brinch Hansen, P.: Multiple-length division revisited: A tour of the minefield. Report 9-1992, Syracuse University (1992), `https://surface.syr.edu/eecs_techreports/166/`
2. Collins, G.E., Musser, D.R.: Analysis of the Pope-Stein division algorithm. Inf. Process. Lett. **6**(5), 151–155 (1977). https://doi.org/10.1016/0020-0190(77)90012-6
3. Gamby, A.N., Katajainen, J.: Convex-hull algorithms: Implementation, testing, and experimentation. Algorithms **11**(12) (2018). https://doi.org/10.3390/a11120195
4. Katajainen, J.: Pure compile-time functions and classes in the CPH MPL. CPH STL report 2017-2, Department of Computer Science, University of Copenhagen (2017), `http://hjemmesider.diku.dk/~jyrki/Myris/Kat2017R.html`
5. Katajainen, J.: Class templates `cphstl::`$\mathbb{N}$ and `cphstl::`$\mathbb{Z}$ for fixed-precision arithmetic. Work in progress (2017–2019)
6. Knuth, D.E.: Seminumerical Algorithms, The Art of Computer Programming, vol. 2. Addison Wesley Longman, 3rd edn. (1998)
7. Krishnamurthy, E.V., Nandi, S.K.: On the normalization requirement of divisor in divide-and-correct methods. Commun. ACM **10**(12), 809–813 (1967). https://doi.org/10.1145/363848.363867
8. Lay-Yong, L.: On the Chinese origin of the galley method of arithmetical division. Br. J. Hist. Sci. **3**(1), 66–69 (1966). https://doi.org/10.1017/S0007087400000200
9. Mifsud, C.J.: A multiple-precision division algorithm. Commun. ACM **13**(11), 666–668 (1970). https://doi.org/10.1145/362790.362795
10. Mifsud, C.J., Bohlen, M.J.: Addendum to a multiple-precision division algorithm. Commun. ACM **16**(10), 628 (1973). https://doi.org/10.1145/362375.362400
11. Pope, D.A., Stein, M.L.: Multiple precision arithmetic. Commun. ACM **3**(12), 652–654 (1960). https://doi.org/10.1145/367487.367499
12. Warren, Jr., H.S.: Hacker's Delight. Pearson Education, Inc., 2nd edn. (2013)
13. Yong, L.L.: The development of Hindu-Arabic and traditional Chinese arithmetic. Chinese Science (13), 35–54 (1996), `https://www.jstor.org/stable/43290379`