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

Jyrki Katajainen

Department of Computer Science, University of Copenhagen

Jyrki Katajainen and Company

More information can be found from my research information system at http://hjemmesider.diku.dk/~jyrki/
History of the standard division algorithm

Galley method

[Sun Tzú, about 400 AD]

[Al-Khwarizmi, 825 AD]

For more information, see [Lay-Yong, 1966] and [Lay Yong, 1996]

Long division

[Briggs, circa 1600 AD]

When performed by hand, different notation is used in different countries; for details, see [Wikipedia 2019]
Main objectives

**Textbook:** Arithmetic Algorithms in Code

**Software package:** multiple-precision arithmetic

**Fast implementation:** Hacker’s Delight [Warren, 2013]

Can I do better?

- discusses a variety of algorithms for common tasks involving integers, often with the aim of performing the minimum number of operations
- Chapter 9: Integer Division
Some implementations

**MIX:** Knuth [1998] (Volume 2)

- 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

**Pascal:** [Brinch Hansen, 1992]

**C:** [Warren, 2013]

**C++:** [this paper]
# Terms

<table>
<thead>
<tr>
<th></th>
<th>General form</th>
<th>Decimal form</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base</strong></td>
<td>$\beta$</td>
<td>10</td>
</tr>
<tr>
<td><strong>Digit</strong></td>
<td>$d_i \in {0, 1, \ldots, \beta - 1}$</td>
<td>$d_i \in {0, 1, \ldots, 9}$</td>
</tr>
<tr>
<td><strong>Number</strong></td>
<td>$d = \langle d_{\ell - 1}, d_{\ell - 2}, \ldots, d_0 \rangle$</td>
<td>string of digits</td>
</tr>
<tr>
<td><strong>Weight of $d_i$</strong></td>
<td>$\beta^i$</td>
<td>$10^i$</td>
</tr>
<tr>
<td><strong>Value of $d$</strong></td>
<td>$\sum_{i=0}^{\ell-1} d_i \cdot \beta^i$</td>
<td></td>
</tr>
</tbody>
</table>


Key observation

Instead of processing the numbers bit by bit, utilize word parallelism!

- [Knuth, 1998]: 16-bit digits
- [Warren, 2013]: 16-bit digits
- [this paper]: Division becomes faster with wider digits!

In general, the division of an $n$-digit number by an $m$-digit number, $n \geq m$, requires $O(m + n + (n - m) \cdot m)$ digit operations.

Q: Which digit width leads to the fastest running time?
Software stack

⨀: one of the integer operations supported by C++, e.g. ==, <, +, −, *, /, % (modulo), <<, >>, ~ (compl), & (bitand), or || (bitor)

⨀(n, m): a function that performs ⨀ when the first operand is an n-digit number and the second operand (if any) an m-digit number

<table>
<thead>
<tr>
<th>Level</th>
<th>Needed operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>⨀(n, m)</td>
<td>/(n, m) (the target), &lt;(n, m)</td>
</tr>
<tr>
<td>⨀(n, n)</td>
<td>⨀ ∈ {==, &lt;, −}</td>
</tr>
<tr>
<td>⨀(n, 1) (ignore overflow)</td>
<td>⨀ ∈ {* , &lt;&lt;}</td>
</tr>
<tr>
<td>⨀(2, 1) (ignore overflow)</td>
<td>⨀ ∈ {+, /}</td>
</tr>
<tr>
<td>⨀(1, 1) (no overflow)</td>
<td>⨀ ∈ {==, &lt;, /, %, &gt;&gt;, &lt;&lt;, &amp;,</td>
</tr>
<tr>
<td>⨀(1, 1) (handle overflow)</td>
<td>⨀ ∈ {+, −, *}</td>
</tr>
<tr>
<td>⨀(1)</td>
<td>⨀ ∈ {~, nlz} (# leading 0 bits)</td>
</tr>
</tbody>
</table>
Example

$k$-digit divisor

\[ 12 \mid 0257835 \]

Quotient

Dividend

Partial remainder

Active part

**Explanation**

\[ 12 \times 0 = 0 \quad \text{//} \quad \ast (k, 1) \]
\[ 2 - 0 = 2 \quad \text{//} \quad - (k, k) \]
\[ 12 \times 2 = 24 \quad \text{//} \quad \ast (k, 1) \]
\[ 25 - 24 = 1 \quad \text{//} \quad - (k, k) \]
\[ 12 \times 1 = 12 \quad \text{//} \quad \ast (k, 1) \]
\[ 17 - 12 = 5 \quad \text{//} \quad - (k, k) \]

**Q:** How to compute a good estimate for the next quotient digit?
Algorithm insight

Normalization: Cast the divisor into the form where its most significant digit is higher than or equal to \([\beta/2]\)

Realization: Multiply both the dividend and the divisor with some factor \(f\), which makes the most significant digit of the divisor large enough [Pope & Stein, 1960]:

\[
\left\lfloor \frac{x \times f}{y \times f} \right\rfloor = \left\lfloor \frac{x}{y} \right\rfloor
\]

Estimation: Use \((2,1)\) with the first two digits of the partial remainder and the most significant digit of the normalized divisor to compute an estimate \(\hat{q}\) for the next quotient digit

Correctness: This estimate is the correct quotient digit, or it is one or two too high [Knuth, 1998] (Theorem B)

Proof by example: For decimal numbers 4 500 and 5\(\ldots\), the estimate is \([45/5] = 9\)

(1) 4 500/501—one correction since 9 \(\times\) 501 = 4 509

(2) 4 500/599—two corrections since 8 \(\times\) 599 = 4 792
Divide $x$ by $y \ (/(n,m))$

**Trivial cases**

1. assert $y \neq 0$  // $\equiv (m,m)$
   
   if $x < y$ return 0  // $\prec (n,m)$

**Space allocation and initialization**

2. Allocate space for the quotient $q = \langle q_{n-m}, q_{n-m-1}, \ldots, q_0 \rangle$
   
   $q \leftarrow 0$

3. Allocate space for the partial remainder $u = \langle u_n, u_{n-1}, \ldots, u_0 \rangle$
   
   $\langle u_{n-1}, u_{n-2}, \ldots, u_0 \rangle \leftarrow x$; $u_n \leftarrow 0$

4. Allocate space for the normalized divisor $v = \langle v_m, v_{m-1}, \ldots, v_0 \rangle$
   
   $\langle v_{m-1}, v_{m-2}, \ldots, v_0 \rangle \leftarrow y$; $v_m \leftarrow 0$

**Normalization**

5. $\sigma \leftarrow \text{nlz} (y_{m-1})$  // Compute the number of leading 0 bits

6. $u \leftarrow u \ll \sigma$  // $\ast (n+1, 1)$ where the multiplier is $2^\sigma$. Since $u$
   
   is one longer than $x$, no overflow is possible

7. $v \leftarrow v \ll \sigma$  // $\ast (m+1, 1)$ where no overflow is possible and
   
   after this the leading bit of $v_{m-1}$ is set
**Main loop**

(8) Compute the digits of \( q \) by letting \( j \) go down from \( n - m \) to 0
   
   (a) \( a = \langle u_{j+m}, u_{j+m-1}, \ldots, u_j \rangle \)  // active part
   
   (b) if \( u_{j+m} \geq v_{m-1} \)
       
       \( \hat{q} \leftarrow \beta - 1 \)

       else
       
       \( \hat{q} \leftarrow \langle u_{j+m}, u_{j+m-1} \rangle / v_{m-1} \)  // /\((2, 1)\)
   
   (c) \( p = \langle p_m, p_{m-1}, \ldots, p_0 \rangle \leftarrow v \ast \hat{q} \)  // *\((m+1, 1)\)
   
   (d) while \( a < p \)  // <\((m+1, m+1)\)
       
       \( \hat{q} \leftarrow \hat{q} - 1 \)

       \( p \leftarrow p - v \)  // −\((m+1, m+1)\)

   (e) \( q_j \leftarrow \hat{q} \)

   \( \langle u_{j+m}, u_{j+m-1}, \ldots, u_j \rangle \leftarrow a - p \)  // −\((m+1, m+1)\)

**Exit**

(9) return \( q \)
Data representation: digits

b: The number of bits in use (specified at compile time)

```cpp
using U = unsigned long long int;
static constexpr std::size_t α = cphmpl::width<U>;
```

- When $0 < b \leq α$, the classes `cphstl::N<b>` are just thin wrappers around the standard unsigned integer types
  ```cpp
  using uints = cphmpl::typelist<
      unsigned char,
      unsigned short int,
      unsigned int,
      unsigned long int,
      unsigned long long int
  >;
  using W = uints::get<cphstl::detail::first_wide_enough<
      uints, b
  >>();
  W data;
  ```

- When $b > α$, a digit is represented as an array of standard integers
  ```cpp
  static constexpr std::size_t n = (b + α - 1) / α; // n = \lceil b/α \rceil
  std::array<U, n> data;
  ```
**Operations: level ⊗(1)**

- `cphstl::leading_zeros`: compute the number of leading 0 bits in the representation of a digit
- `cphstl::some_trailing_ones`: generate a digit having a specific number of trailing 1 bits in its representation
- `cphstl::detail::lower_half` & `cphstl::detail::upper_half`: get from a digit its two halves

- These functions are overloaded to work differently depending on the type of the argument
- For the standard integer types, it can call an intrinsic function that will be translated into a single hardware instruction
- There are also `constexpr` forms that compute the result at compile time if the argument is known at that time
Operations: level \( (1, 1) \)

**Standard (unsigned) integers**

- `unsigned char`: 8 bits
- `unsigned short`: 16 bits
- `unsigned int`: 32 bits
- `unsigned long`: 64 bits (Linux)

**CPH STL (unsigned) integers**

- `cphstl::\mathbb{N}<b>`: \( b \) bits (for any \( b > 0 \))

**Overflow handling for \( \{+,-,\ast\} \)**

Described in Warren’s book
Ideas: Taken from Warren’s book, e.g. \(/(2,1)\)

Contribution: Generic programming, metaprogramming

```cpp
template <typename D, typename W>
requires
/* 1 */ cphmpl::is_unsigned<W> and
/* 2 */ std::is_same_v<D, cphmpl::twice_wider<W>> and
/* 3 */ cphstl::detail::uints::is_member<D>
constexpr D divide(D const & x, W const & y) {
    D u = static_cast<D>(y);
    D z = x / u; // /(1,1)
    return z;
}

template <typename D, typename W>
requires
/* 1 */ cphmpl::is_unsigned<W> and
/* 2 */ std::is_same_v<D, cphmpl::twice_wider<W>> and
/* 3 */ not cphstl::detail::uints::is_member<D>
constexpr D divide(D const & x, W const & y) {
    W x0 = cphstl::detail::lower_half<W>(x);
    W x1 = cphstl::detail::upper_half<W>(x);
    W q1 = x1 / y; // /(1,1)
    W u1 = x1 % y; // %/(1,1)
    W q0 = cphstl::detail::divide_long_unsigned(x0, u1, y);
    D q = cphstl::detail::halves_together<D>(q0, q1);
    return q;
}
```
Data representation: numbers

- The digit strings 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 string. 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.
Operations: level $\odot(n, 1)$

```cpp
template <typename L, typename R, typename W>
requires
/* 1 */ cphmpl::specifies_range<L> and
/* 2 */ cphmpl::specifies_range<R> and
/* 3 */ cphmpl::is_unsigned<W> and
/* 4 */ std::is_same_v<cphmpl::value<L>, W> and
/* 5 */ std::is_same_v<cphmpl::value<R>, W>
void product(L& result, R const& multiplicand, W const& factor) {
  assert(std::size(result) == std::size(multiplicand));
  using D = cphmpl::twice_wider<W>;
  using I = cphmpl::iterator<L>;
  using J = cphmpl::const_iterator<R>;
  J first = std::cbegin(multiplicand);
  J past = std::cend(multiplicand);
  W carry = W();
  I q = std::begin(result);
  for (J p = first; p != past; ++p, ++q) {
    D t = cphstl::detail::multiply<D>(*p, factor); // *(1,1)
    t = cphstl::detail::add(t, carry); // +(2,1)
    *q = cphstl::detail::lower_half<W>(t);
    carry = cphstl::detail::upper_half<W>(t);
  }
}
```
Operations: level \( \odot (n, n) \)

```
template<typename L, typename R>
requires
/* 1 */ cphmpl::specifies_range<L> and
/* 2 */ cphmpl::specifies_range<R> and
/* 3 */ std::is_same_v<cphmpl::value<L>, cphmpl::value<R>>
bool is_less(L const & lhs, R const & rhs) {
    // check whether lhs < rhs or not
    assert(std::size(lhs) == std::size(rhs));
    assert(!std::empty(lhs));
    using I = cphmpl::const_iterator<L>;
    using J = cphmpl::const_iterator<R>;
    I p = std::cend(lhs);
    J q = std::cend(rhs);
    I first = std::cbegin(lhs);
    do {
    --p;
    --q;
    } while (p != first and *p == *q); // == (1, 1)
    return *p < *q; // < (1, 1)
}
```

Inner loop in assembler for 8-byte digits

```assembly
.L2:
movq (%rdx), %rsi
cmpq %rsi, (%rax)
jne .L3
subq $8, %rax
subq $8, %rdx
cmpq %rdi, %rax
jne .L2
.L3:
```
Intel cost of the critical subroutines

Set-up:

\(<(n, n)\): \texttt{is\_less} compared two equal numbers

\(-(n, n)\): \texttt{difference} processed two random numbers, except that the first was made larger by resetting the most significant digits

\(*(n, 1)\): \texttt{product} multiplied a random number with a random digit

Performance indicator: \# instructions executed per digit

Tools: \texttt{perf stat} (performance analyser); \texttt{g++} (compiler)

<table>
<thead>
<tr>
<th>Width</th>
<th>\texttt{is_less}</th>
<th>\texttt{difference}</th>
<th>\texttt{product}</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.17</td>
<td>12.30</td>
<td>9.17</td>
</tr>
<tr>
<td>16</td>
<td>7.16</td>
<td>14.28</td>
<td>10.16</td>
</tr>
<tr>
<td>32</td>
<td>7.16</td>
<td>14.24</td>
<td>10.15</td>
</tr>
<tr>
<td>64</td>
<td>7.18</td>
<td>15.24</td>
<td>26.17</td>
</tr>
<tr>
<td>128</td>
<td>10.40</td>
<td>33.51</td>
<td>6.39</td>
</tr>
<tr>
<td>256</td>
<td>16.68</td>
<td>71.88</td>
<td>6.67</td>
</tr>
<tr>
<td>512</td>
<td>29.26</td>
<td>148.62</td>
<td>2,874</td>
</tr>
<tr>
<td>1024</td>
<td>54.37</td>
<td>317.04</td>
<td>14,339</td>
</tr>
</tbody>
</table>
## Experimental results: small numbers

**Set-up:** Perform scalar-vector arithmetic for different digit types

**Performance indicator:** # instructions executed per operation

<table>
<thead>
<tr>
<th>Type</th>
<th>+</th>
<th>−</th>
<th>*</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char</td>
<td>0.40</td>
<td>0.40</td>
<td>0.90</td>
<td>6.02</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>0.46</td>
<td>0.46</td>
<td>0.46</td>
<td>4.53</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0.89</td>
<td>0.89</td>
<td>2.39</td>
<td>4.52</td>
</tr>
<tr>
<td>unsigned long long int</td>
<td>1.77</td>
<td>1.77</td>
<td>5.77</td>
<td>4.53</td>
</tr>
<tr>
<td>unsigned __int128</td>
<td>5.53</td>
<td>5.53</td>
<td>7.04</td>
<td>19.55</td>
</tr>
<tr>
<td>cphstl :: N&lt;8&gt;</td>
<td>0.25</td>
<td>0.25</td>
<td>0.72</td>
<td>6.02</td>
</tr>
<tr>
<td>cphstl :: N&lt;16&gt;</td>
<td>0.46</td>
<td>0.46</td>
<td>0.46</td>
<td>7.02</td>
</tr>
<tr>
<td>cphstl :: N&lt;24&gt;</td>
<td>1.14</td>
<td>1.14</td>
<td>2.77</td>
<td>7.02</td>
</tr>
<tr>
<td>cphstl :: N&lt;32&gt;</td>
<td>0.89</td>
<td>0.89</td>
<td>2.39</td>
<td>7.02</td>
</tr>
<tr>
<td>cphstl :: N&lt;48&gt;</td>
<td>2.27</td>
<td>2.27</td>
<td>6.27</td>
<td>7.03</td>
</tr>
<tr>
<td>cphstl :: N&lt;64&gt;</td>
<td>1.77</td>
<td>1.77</td>
<td>5.77</td>
<td>7.03</td>
</tr>
<tr>
<td>cphstl :: N&lt;128&gt;</td>
<td>5.54</td>
<td>7.54</td>
<td>24.07</td>
<td>18.12</td>
</tr>
<tr>
<td>cphstl :: N&lt;256&gt;</td>
<td>18.06</td>
<td>27.06</td>
<td>161.9</td>
<td>49.37</td>
</tr>
<tr>
<td>cphstl :: N&lt;512&gt;</td>
<td>38.11</td>
<td>81.11</td>
<td>407.6</td>
<td>73.61</td>
</tr>
<tr>
<td>cphstl :: N&lt;1024&gt;</td>
<td>96.21</td>
<td>179.2</td>
<td>1396</td>
<td>129.3</td>
</tr>
</tbody>
</table>
Experimental results: large numbers

Set-up: Run the long-division program for two random numbers of $N$ and $\frac{N}{2}$ bits

Performance indicator: # instructions executed for different digit widths; the values indicate the coefficient $C$ in the formula $C \cdot \left(\frac{N}{64}\right)^2$

<table>
<thead>
<tr>
<th>Width</th>
<th>$N = 2^{12}$</th>
<th>$N = 2^{14}$</th>
<th>$N = 2^{16}$</th>
<th>$N = 2^{18}$</th>
<th>$N = 2^{20}$</th>
<th>$N = 2^{22}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>447.1</td>
<td>390.5</td>
<td>427.2</td>
<td>361.9</td>
<td>410.5</td>
<td>392.7</td>
</tr>
<tr>
<td>16</td>
<td>110.9</td>
<td>97.1</td>
<td>124.1</td>
<td>113.9</td>
<td>104.6</td>
<td>135.0</td>
</tr>
<tr>
<td>32</td>
<td>34.2</td>
<td>32.1</td>
<td>28.6</td>
<td>30.8</td>
<td>26.6</td>
<td>29.1</td>
</tr>
<tr>
<td>64</td>
<td>14.4</td>
<td>12.5</td>
<td>10.9</td>
<td>12.6</td>
<td>11.9</td>
<td>11.3</td>
</tr>
<tr>
<td>128</td>
<td>5.8</td>
<td>3.2</td>
<td>2.6</td>
<td>2.4</td>
<td>2.4</td>
<td>2.4</td>
</tr>
<tr>
<td>256</td>
<td>13.5</td>
<td>4.3</td>
<td>1.9</td>
<td>1.3</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>512</td>
<td>15.5</td>
<td>3.8</td>
<td>1.3</td>
<td>0.7</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>1024</td>
<td>36.1</td>
<td>18.5</td>
<td>14.7</td>
<td>13.9</td>
<td>13.7</td>
<td>13.6</td>
</tr>
</tbody>
</table>

| 16 *)  | 63.3         | 60.8         | 60.2         | 60.0         | 60.0         | 60.0         |

*) [Warren, 2013] (Figure 9-3)
Final remarks

**Memory efficiency:** \(\text{sizeof}(\text{cphstl}::\mathbb{N}<64>) = 8\), i.e. there is no space overhead, whereas a dynamic solution must store the length and allocate space for the digits dynamically.

**Application efficiency:** Test the library facilities in real applications [Referee]; here I rely on crowd sourcing!

**Politics:** Push \(\text{cphstl}::\mathbb{N}\) and \(\text{cphstl}::\mathbb{Z}\) class templates to the C++ standard library.

**Further research:** Devise a division algorithm that is asymptotically faster and practical.
Software overview (June 2019)

In the lines-of-code (LOC) counts (1) all comments, (2) lines only having a single parenthesis, (3) debugging aids, and (4) assertions are excluded.

Warren’s division program

<table>
<thead>
<tr>
<th>File</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>divmnu.c++</td>
<td>91</td>
</tr>
</tbody>
</table>

Metaprogramming package

<table>
<thead>
<tr>
<th>File</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>cphmpl/charlist.h++</td>
<td>40</td>
</tr>
<tr>
<td>cphmpl/functions.h++</td>
<td>501</td>
</tr>
<tr>
<td>cphmpl/intlist.h++</td>
<td>25</td>
</tr>
<tr>
<td>cphmpl/lists.h++</td>
<td>5</td>
</tr>
<tr>
<td>cphmpl/typelist.h++</td>
<td>156</td>
</tr>
<tr>
<td>cphmpl/valuelist.h++</td>
<td>159</td>
</tr>
</tbody>
</table>

Integer package

<table>
<thead>
<tr>
<th>File</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>cphstl/bit-tricks.h++</td>
<td>274</td>
</tr>
<tr>
<td>cphstl/constants.h++</td>
<td>347</td>
</tr>
<tr>
<td>cphstl/integers.h++</td>
<td>2628</td>
</tr>
<tr>
<td>cphstl/math.h++</td>
<td>78</td>
</tr>
<tr>
<td>cphstl/ranges.h++</td>
<td>62</td>
</tr>
</tbody>
</table>