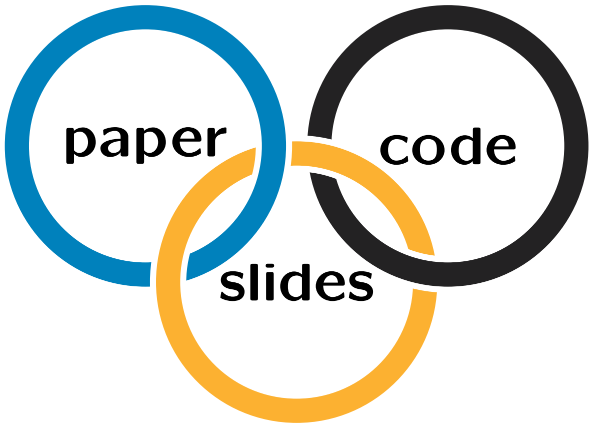
29 June, 2019 SEA^2, Kalamata; Last revision: 1 August, 2019

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]

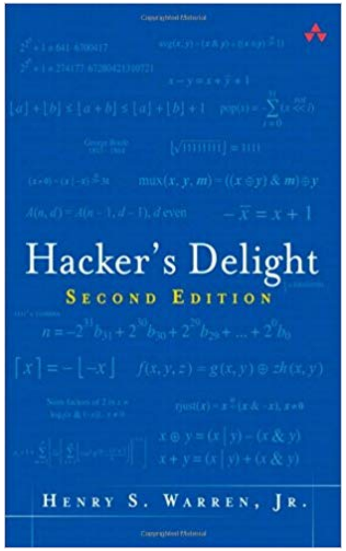
When performed by hand, different notation is used in different countries; for details, see [Wikipedia 2019]

Main objectives

Textbook: *Arithmetic Algorithms in Code*

Software package: multiple-precision arithmetic

Fast implementation: *Hacker's Delight* [Warren, 2013]
    Can I do better?

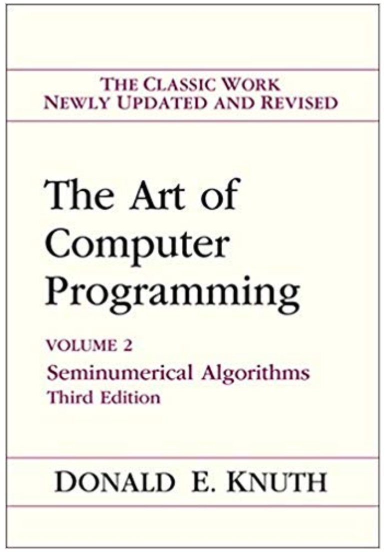


- discusses a variety of algorithms for common tasks involving integers, often with the aim of performing the minimum number of operations
- Chapter 9: Integer Division

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

	General form	Decimal form
Base	β	10
Digit	$d_i \in \{0, 1, \ldots, \beta - 1\}$	$d_i \in \{0, 1, \ldots, 9\}$
Number	$\boldsymbol{d} = \langle d_{\ell-1}, d_{\ell-2}, \ldots, d_0 \rangle$	string of digits
Weight of d_i	β^i	10^i
Value of $\boldsymbol{d}$	$\displaystyle\sum_{i=0}^{\ell-1} d_i \cdot \beta^i$	

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?

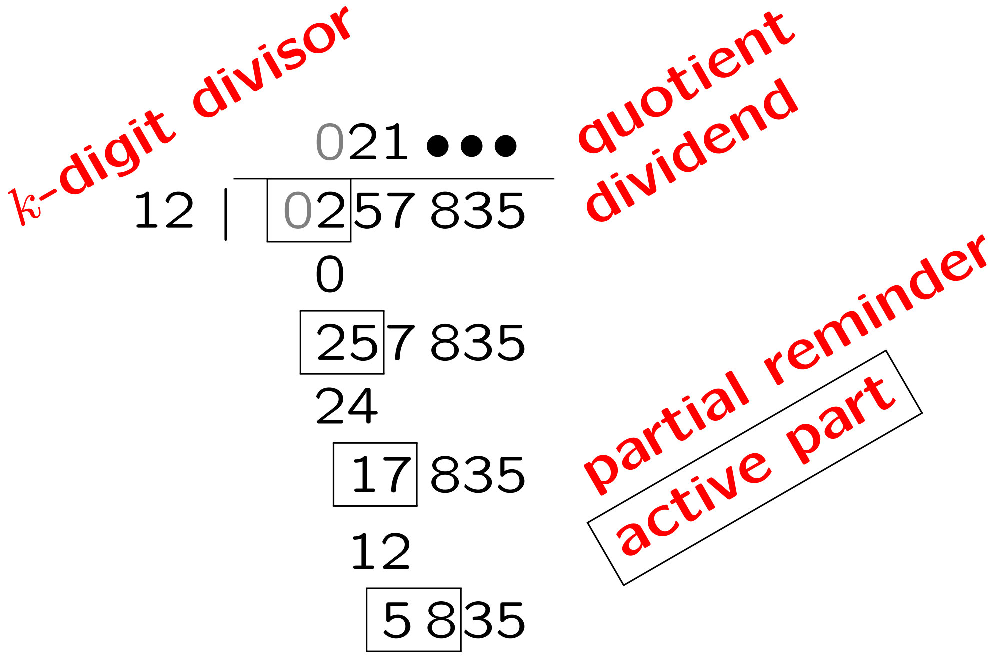
Software stack

$\odot$: one of the integer operations supported by C++, e.g. $==$, $<$, $+$, $-$, $*$, $/$, % (modulo), $<<$, $>>$, $\sim$ (**compl**), & (**bitand**), or $||$ (**bitor**)

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

Level	Needed operations
$\odot(\boldsymbol{n}, \boldsymbol{m})$	$/(\boldsymbol{n}, \boldsymbol{m})$ (the target), $<(\boldsymbol{n}, \boldsymbol{m})$
$\odot(\boldsymbol{n}, \boldsymbol{n})$	$\odot \in \{==, <, -\}$
$\odot(\boldsymbol{n}, \mathbf{1})$ (ignore overflow)	$\odot \in \{*, <<\}$
$\odot(\mathbf{2}, \mathbf{1})$ (ignore overflow)	$\odot \in \{+, /\}$
$\odot(\mathbf{1}, \mathbf{1})$ (no overflow)	$\odot \in \{==, <, /, \%, >>, <<, \&, \|\|\}$
$\odot(\mathbf{1}, \mathbf{1})$ (handle overflow)	$\odot \in \{+, -, *\}$
$\odot(\mathbf{1})$	$\odot \in \{\sim, \mathbf{nlz}\}$ (# leading 0 bits)

Example



Explanation

$$
\begin{array}{ll}
12 * 0 = 0 & // \ *(k,1) \\
2 - 0 = 2 & // -(k,k) \\
12 * 2 = 24 & // \ *(k,1) \\
25 - 24 = 1 & // -(k,k) \\
12 * 1 = 12 & // \ *(k,1) \\
17 - 12 = 5 & // -(k,k)
\end{array}
$$

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 $\lfloor \beta/2 \rfloor$

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]: $\left\lfloor \dfrac{\boldsymbol{x} * f}{\boldsymbol{y} * f} \right\rfloor = \left\lfloor \dfrac{\boldsymbol{x}}{\boldsymbol{y}} \right\rfloor$

Estimation: Use $/(\mathbf{2}, \mathbf{1})$ with the first two digits of the partial reminder and the most significant digit of the normalized divisor to compute an estimate $\widehat{\mathrm{q}}$ 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••, the estimate is $\lfloor 45/5 \rfloor = 9$

(1) 4 500/501—one correction since $9 * 501 = 4\,509$

(2) 4 500/599—two corrections since $8 * 599 = 4\,792$

Divide x by y $(/(n, m))$

Trivial cases

(1) **assert** $y \neq 0$ // $==(m, m)$
if $x < y$ **return** 0 // $<(n, m)$

Space allocation and initialization

(2) Allocate space for the quotient $\boldsymbol{q} = \langle q_{n-m}, q_{n-m-1}, \ldots, q_0 \rangle$
$\boldsymbol{q} \leftarrow \mathbf{0}$
(3) Allocate space for the partial remainder $\boldsymbol{u} = \langle u_n, u_{n-1}, \ldots, u_0 \rangle$
$\langle u_{n-1}, u_{n-2}, \ldots, u_0 \rangle \leftarrow \boldsymbol{x}$; $u_n \leftarrow 0$
(4) Allocate space for the normalized divisor $\boldsymbol{v} = \langle v_m, v_{m-1}, \ldots, v_0 \rangle$
$\langle v_{m-1}, v_{m-2}, \ldots, v_0 \rangle \leftarrow \boldsymbol{y}$; $v_m \leftarrow 0$

Normalization

(5) $\sigma \leftarrow \mathbf{nlz}(y_{m-1})$ // Compute the number of leading 0 bits
(6) $\boldsymbol{u} \leftarrow \boldsymbol{u} << \sigma$ // $*(\boldsymbol{n}+\mathbf{1}, \mathbf{1})$ where the multiplier is 2^σ. Since $\boldsymbol{u}$ is one longer than $\boldsymbol{x}$, no overflow is possible
(7) $\boldsymbol{v} \leftarrow \boldsymbol{v} << \sigma$ // $*(\boldsymbol{m}+\mathbf{1}, \mathbf{1})$ where no overflow is possible and after this the leading bit of v_{m-1} is set

Main loop

(8) Compute the digits of $\boldsymbol{q}$ by letting j go down from $n-m$ to 0

(a) $\boldsymbol{a} = \left\langle u_{j+m}, u_{j+m-1}, \ldots, u_j \right\rangle$ // active part

(b) **if** $u_{j+m} \geq v_{m-1}$

$\hat{\mathsf{q}} \leftarrow \beta - 1$

else

$\hat{\mathsf{q}} \leftarrow \left\langle u_{j+m}, u_{j+m-1} \right\rangle / v_{m-1}$ // $/\mathbf{(2,1)}$

(c) $\boldsymbol{p} = \langle p_m, p_{m-1}, \ldots, p_0 \rangle \leftarrow \boldsymbol{v} * \hat{\mathsf{q}}$ // $\boldsymbol{*(m+1,1)}$

(d) **while** $\boldsymbol{a} < \boldsymbol{p}$ // $\boldsymbol{<(m+1,m+1)}$

$\hat{\mathsf{q}} \leftarrow \hat{\mathsf{q}} - 1$

$\boldsymbol{p} \leftarrow \boldsymbol{p} - \boldsymbol{v}$ // $\boldsymbol{-(m+1,m+1)}$

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

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

critical subroutines

Exit

(9) **return** $\boldsymbol{q}$

Data representation: digits

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

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

constraint-based overloading

- When $0 < \mathtt{b} \leq \alpha$, the classes cphstl::ℕ<b> are just thin wrappers around the standard unsigned integer types

```
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 $\mathtt{b} > \alpha$, a digit is represented as an array of standard integers

```
static constexpr std::size_t n = (b + α − 1) / α; // n = ⌈b/α⌉
std::array<U, n> data;
```

Operations: level $\odot(1)$

`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 $\bigodot(1,1)$

Standard (unsigned) integers

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

CPH STL (unsigned) integers

$\texttt{cphstl} :: \mathbb{N}{<}\texttt{b}{>}$ $\quad$ $\texttt{b}$ bits (for any $\texttt{b} > 0$)

Overflow handling for $\{+,\ -,\ *\}$

Described in Warren's book

Operations: level $\odot(2,1)$

Ideas: Taken from Warren's book, e.g. $/(2,1)$

Contribution: Generic programming, metaprogramming

```
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 $\odot(n, 1)$

```
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) {
  // compute result = multiplicand * 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(not 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

```
.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:

$<(n, n)$: `is_less` compared two equal numbers

$-(n, n)$: `difference` processed two random numbers, except that the first was made larger by resetting the most significant digits

$*(n, 1)$: `product` multiplied a random number with a random digit

Performance indicator: # instructions executed per digit

Tools: `perf stat` (performance analyser); `g++` (compiler)

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

hardware support

Experimental results: small numbers

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

Performance indicator: # instructions executed per operation

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 :: $\mathbb{N}<8>$	0.25	0.25	0.72	6.02
cphstl :: $\mathbb{N}<16>$	0.46	0.46	0.46	7.02
cphstl :: $\mathbb{N}<24>$	1.14	1.14	2.77	7.02
cphstl :: $\mathbb{N}<32>$	0.89	0.89	2.39	7.02
cphstl :: $\mathbb{N}<48>$	2.27	2.27	6.27	7.03
cphstl :: $\mathbb{N}<64>$	1.77	1.77	5.77	7.03
cphstl :: $\mathbb{N}<128>$	5.54	7.54	24.07	18.12
cphstl :: $\mathbb{N}<256>$	18.06	27.06	161.9	49.37
cphstl :: $\mathbb{N}<512>$	38.11	81.11	407.6	73.61
cphstl :: $\mathbb{N}<1024>$	96.21	179.2	1396	129.3

quadratic

hardware support

division by 0 sanitation

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 $C \cdot \left(\frac{N}{64}\right)^2$

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 $^{\star)}$	63.3	60.8	60.2	60.0	60.0	60.0

$^{\star)}$ [Warren, 2013] (Figure 9-3)

Final remarks

Memory efficiency: $\text{sizeof}(\texttt{cphstl}::\mathbb{N}{<}64{>}) = 8$, 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 $\texttt{cphstl}::\mathbb{N}$ and $\texttt{cphstl}::\mathbb{Z}$ 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

File	LOC
`divmnu.c++`	91

Metaprogramming package

File	LOC
`cphmpl/charlist.h++`	40
`cphmpl/functions.h++`	501
`cphmpl/intlist.h++`	25
`cphmpl/lists.h++`	5
`cphmpl/typelist.h++`	156
`cphmpl/valuelist.h++`	159

Integer package

File	LOC
`cphstl/bit-tricks.h++`	274
`cphstl/constants.h++`	347
`cphstl/integers.h++`	2 628
`cphstl/math.h++`	78
`cphstl/ranges.h++`	62