# Division algorithm

A **division algorithm** is an algorithm which, given two integers $N$ and $D$ (respectively the numerator and the denominator), computes their quotient and/or remainder, the result of Euclidean division. Some are applied by hand, while others are employed by digital circuit designs and software.

Division algorithms fall into two main categories: slow division and fast division. Slow division algorithms produce one digit of the final quotient per iteration. Examples of slow division include restoring, non-performing restoring, non-restoring, and SRT division. Fast division methods start with a close approximation to the final quotient and produce twice as many digits of the final quotient on each iteration.[1] Newton–Raphson and Goldschmidt algorithms fall into this category.

Variants of these algorithms allow using fast multiplication algorithms. It results that, for large integers, the computer time needed for a division is the same, up to a constant factor, as the time needed for a multiplication, whichever multiplication algorithm is used.

Discussion will refer to the form $N/D = (Q, R)$, where

- $N$ = numerator (dividend)
- $D$ = denominator (divisor)

is the input, and

- $Q$ = quotient
- $R$ = remainder

is the output.

## Division by repeated subtraction

The simplest division algorithm, historically incorporated into a greatest common divisor algorithm presented in Euclid's *Elements*, Book VII, Proposition 1, finds the remainder given two positive integers using only subtractions and comparisons:

```
R := N
Q := 0
while R ≥ D do
  R := R − D
  Q := Q + 1
end
return (Q,R)
```

The proof that the quotient and remainder exist and are unique (described at Euclidean division) gives rise to a complete division algorithm, applicable to both negative and positive numbers, using additions, subtractions, and comparisons:

```
function divide(N, D)
  if D = 0 then error(DivisionByZero) end
  if D < 0 then (Q, R) := divide(N, −D); return (−Q, R) end
  if N < 0 then
    (Q,R) := divide(−N, D)
    if R = 0 then return (−Q, 0)
    else return (−Q − 1, D − R) end
  end
  -- At this point, N ≥ 0 and D > 0
  return divide_unsigned(N, D)
end
function divide_unsigned(N, D)
  Q := 0; R := N
  while R ≥ D do
    Q := Q + 1
    R := R − D
  end
  return (Q, R)
end
```

This procedure always produces R ≥ 0. Although very simple, it takes Ω(Q) steps, and so is exponentially slower than even slow division algorithms like long division. It is useful if Q is known to be small (being an output-sensitive algorithm), and can serve as an executable specification.

## Long division

Long division is the standard algorithm used for pen-and-paper division of multi-digit numbers expressed in decimal notation. It shifts gradually from the left to the right end of the dividend, subtracting the largest possible multiple of the divisor (at the digit level) at each stage; the multiples then become the digits of the quotient, and the final difference is then the remainder.

When used with a binary radix, this method forms the basis for the (unsigned) integer division with remainder algorithm below. Short division is an abbreviated form of long division suitable for one-digit divisors. Chunking – also known as the partial quotients method or the hangman method – is a less-efficient form of long division which may be easier to understand. By allowing one to subtract more multiples than what one currently has at each stage, a more freeform variant of long division can be developed as well.

### Integer division (unsigned) with remainder

The following algorithm, the binary version of the famous long division, will divide *N* by *D*, placing the quotient in *Q* and the remainder in *R*. In the following pseudo-code, all values are treated as unsigned integers.

```
if D = 0 then error(DivisionByZeroException) end
Q := 0                  -- Initialize quotient and remainder to zero
R := 0
for i := n − 1 .. 0 do  -- Where n is number of bits in N
  R := R << 1           -- Left-shift R by 1 bit
  R(0) := N(i)          -- Set the least-significant bit of R equal to bit i of the numerator
  if R ≥ D then
    R := R − D
    Q(i) := 1
  end
end
```

#### Example

If we take N=$1100_2$ ($12_{10}$) and D=$100_2$ ($4_{10}$)

*Step 1*: Set R=0 and Q=0
*Step 2*: Take i=3 (one less than the number of bits in N)
*Step 3*: R=00 (left shifted by 1)
*Step 4*: R=01 (setting R(0) to N(i))
*Step 5*: R < D, so skip statement

*Step 2*: Set i=2
*Step 3*: R=010
*Step 4*: R=011
*Step 5*: R < D, statement skipped

*Step 2*: Set i=1
*Step 3*: R=0110
*Step 4*: R=0110
*Step 5*: R>=D, statement entered
*Step 5b*: R=10 (R−D)
*Step 5c*: Q=10 (setting Q(i) to 1)

*Step 2*: Set i=0
*Step 3*: R=100
*Step 4*: R=100
*Step 5*: R>=D, statement entered
*Step 5b*: R=0 (R−D)
*Step 5c*: Q=11 (setting Q(i) to 1)

**end**
Q=$11_2$ ($3_{10}$) and R=0.

## Slow division methods

Slow division methods are all based on a standard recurrence equation [2]

$$R_{j+1} = B \times R_j - q_{n-(j+1)} \times D,$$

where:

- $R_j$ is the $j$-th partial remainder of the division
- $B$ is the radix (base, usually 2 internally in computers and calculators)
- $q_{n-(j+1)}$ is the digit of the quotient in position $n-(j+1)$, where the digit positions are numbered from least-significant 0 to most significant $n-1$
- $n$ is number of digits in the quotient
- $D$ is the divisor

### Restoring division

Restoring division operates on fixed-point fractional numbers and depends on the assumption $0 < D < N$.

The quotient digits $q$ are formed from the digit set {0,1}.

The basic algorithm for binary (radix 2) restoring division is:

```
R := N
D := D << n            -- R and D need twice the word width of N and Q
for i := n − 1 .. 0 do  -- For example 31..0 for 32 bits
  R := 2 * R − D          -- Trial subtraction from shifted value (multiplication by 2 is a shift in binary representation)
  if R >= 0 then
    q(i) := 1          -- Result-bit 1
  else
    q(i) := 0          -- Result-bit 0
    R := R + D         -- New partial remainder is (restored) shifted value
  end
end

-- Where: N = numerator, D = denominator, n = #bits, R = partial remainder, q(i) = bit #i of quotient
```

Non-performing restoring division is similar to restoring division except that the value of 2R is saved, so $D$ does not need to be added back in for the case of R < 0.

## Non-restoring division

Non-restoring division uses the digit set {−1, 1} for the quotient digits instead of {0, 1}. The algorithm is more complex, but has the advantage when implemented in hardware that there is only one decision and addition/subtraction per quotient bit; there is no restoring step after the subtraction,[3] which potentially cuts down the numbers of operations by up to half and lets it be executed faster.[4] The basic algorithm for binary (radix 2) non-restoring division of non-negative numbers is:

```
R := N
D := D << n            -- R and D need twice the word width of N and Q
for i = n − 1 .. 0 do  -- for example 31..0 for 32 bits
  if R >= 0 then
    q(i) := +1
    R := 2 * R − D
  else
    q(i) := −1
    R := 2 * R + D
  end if
end

-- Note: N=numerator, D=denominator, n=#bits, R=partial remainder, q(i)=bit #i of quotient.
```

Following this algorithm, the quotient is in a non-standard form consisting of digits of −1 and +1. This form needs to be converted to binary to form the final quotient. Example:

Convert the following quotient to the digit set {0,1}:

| | |
|---|---|
| Start: | $Q = 111\bar{1}1\bar{1}1\bar{1}$ |
| 1. Form the positive term: | $P = 11101010$ |
| 2. Mask the negative term*: | $M = 00010101$ |
| 3. Subtract: $P - M$: | $Q = 11010101$ |

*.( Signed binary notation with ones' complement without two's complement)

If the −1 digits of $Q$ are stored as zeros (0) as is common, then $P$ **is** $Q$ and computing $M$ is trivial: perform a ones' complement (bit by bit complement) on the original $Q$.

```
Q := Q − bit.bnot(Q)      -- Appropriate if −1 digits in Q are represented as zeros as is common.
```

Finally, quotients computed by this algorithm are always odd, and the remainder in R is in the range −D ≤ R < D. For example, 5 / 2 = 3 R −1. To convert to a positive remainder, do a single restoring step *after* Q is converted from non-standard form to standard form:

```
if R < 0 then
  Q := Q − 1
  R := R + D  -- Needed only if the remainder is of interest.
end if
```

The actual remainder is R >> n. (As with restoring division, the low-order bits of R are used up at the same rate as bits of the quotient Q are produced, and it is common to use a single shift register for both.)

## SRT division

SRT division is a popular method for division in many microprocessor implementations.[5][6] The algorithm is named after D. W. Sweeney of IBM, James E. Robertson of University of Illinois, and K. D. Tocher of Imperial College London. They all developed the algorithm independently at approximately the same time (published in February 1957, September 1958, and January 1958 respectively).[7][8][9]

SRT division is similar to non-restoring division, but it uses a lookup table based on the dividend and the divisor to determine each quotient digit.

The most significant difference is that a *redundant representation* is used for the quotient. For example, when implementing radix-4 SRT division, each quotient digit is chosen from *five* possibilities: { −2, −1, 0, +1, +2 }. Because of this, the choice of a quotient digit need not be perfect; later quotient digits can correct for slight errors. (For example, the quotient digit pairs (0, +2) and (1, −2) are equivalent, since 0×4+2 = 1×4−2.) This tolerance allows quotient digits to be selected using only a few most-significant bits of the dividend and divisor, rather than requiring a full-width subtraction. This simplification in turn allows a radix higher than 2 to be used.

Like non-restoring division, the final steps are a final full-width subtraction to resolve the last quotient bit, and conversion of the quotient to standard binary form.

The Intel Pentium processor's infamous floating-point division bug was caused by an incorrectly coded lookup table. Five of the 1066 entries had been mistakenly omitted.[10][11]

# Fast division methods

### Newton–Raphson division

Newton–Raphson uses Newton's method to find the reciprocal of $D$ and multiply that reciprocal by $N$ to find the final quotient $Q$.

The steps of Newton–Raphson division are:

1. Calculate an estimate $X_0$ for the reciprocal $1/D$ of the divisor $D$.
2. Compute successively more accurate estimates $X_1, X_2, \ldots, X_S$ of the reciprocal. This is where one employs the Newton–Raphson method as such.
3. Compute the quotient by multiplying the dividend by the reciprocal of the divisor: $Q = NX_S$.

In order to apply Newton's method to find the reciprocal of $D$, it is necessary to find a function $f(X)$ that has a zero at $X = 1/D$. The obvious such function is $f(X) = DX - 1$, but the Newton–Raphson iteration for this is unhelpful, since it cannot be computed without already knowing the reciprocal of $D$ (moreover it attempts to compute the exact reciprocal in one step, rather than allow for iterative improvements). A function that does work is $f(X) = (1/X) - D$, for which the Newton–Raphson iteration gives

$$X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)} = X_i - \frac{1/X_i - D}{-1/X_i^2} = X_i + X_i(1 - DX_i) = X_i(2 - DX_i),$$

which can be calculated from $X_i$ using only multiplication and subtraction, or using two fused multiply–adds.

From a computation point of view, the expressions $X_{i+1} = X_i + X_i(1 - DX_i)$ and $X_{i+1} = X_i(2 - DX_i)$ are not equivalent. To obtain a result with a precision of 2*n* bits while making use of the second expression, one must compute the product between $X_i$ and $(2 - DX_i)$ with double the given precision of $X_i$(*n* bits). In contrast, the product between $X_i$ and $(1 - DX_i)$ need only be computed with a precision of *n* bits, because the leading *n* bits (after the binary point) of $(1 - DX_i)$ are zeros.

If the error is defined as $\varepsilon_i = 1 - DX_i$, then:

$$\begin{aligned}\varepsilon_{i+1} &= 1 - DX_{i+1} \\ &= 1 - D(X_i(2 - DX_i)) \\ &= 1 - 2DX_i + D^2X_i^2 \\ &= (1 - DX_i)^2 \\ &= {\varepsilon_i}^2.\end{aligned}$$

This squaring of the error at each iteration step – the so-called quadratic convergence of Newton–Raphson's method – has the effect that the number of correct digits in the result roughly *doubles for every iteration*, a property that becomes extremely valuable when the numbers involved have many digits (e.g. in the large integer domain). But it also means that the initial convergence of the method can be comparatively slow, especially if the initial estimate $X_0$ is poorly chosen.

For the subproblem of choosing an initial estimate $X_0$, it is convenient to apply a bit-shift to the divisor *D* to scale it so that 0.5 ≤ *D* ≤ 1; by applying the same bit-shift to the numerator *N*, one ensures the quotient does not change. Then one could use a linear approximation in the form

$$X_0 = T_1 + T_2 D \approx \frac{1}{D}$$

to initialize Newton–Raphson. To minimize the maximum of the absolute value of the error of this approximation on interval $[0.5, 1]$, one should use

$$X_0 = \frac{48}{17} - \frac{32}{17} D.$$

The coefficients of the linear approximation are determined as follows. The absolute value of the error is $|\varepsilon_0| = |1 - D(T_1 + T_2 D)|$. The minimum of the maximum absolute value of the error is determined by the Chebyshev equioscillation theorem applied to $F(D) = 1 - D(T_1 + T_2 D)$. The local minimum of $F(D)$ occurs when $F'(D) = 0$, which has solution $D = -T_1/(2T_2)$. The function at that minimum must be of opposite sign as the function at the endpoints, namely, $F(1/2) = F(1) = -F(-T_1/(2T_2))$. The two equations in the two unknowns have a unique solution $T_1 = 48/17$ and $T_2 = -32/17$, and the maximum error is $F(1) = 1/17$. Using this approximation, the absolute value of the error of the initial value is less than

$$|\varepsilon_0| \leq \frac{1}{17} \approx 0.059.$$

It is possible to generate a polynomial fit of degree larger than 1, computing the coefficients using the Remez algorithm. The trade-off is that the initial guess requires more computational cycles but hopefully in exchange for fewer iterations of Newton–Raphson.

Since for this method the convergence is exactly quadratic, it follows that

$$S = \left\lceil \log_2 \frac{P+1}{\log_2 17} \right\rceil$$

steps are enough to calculate the value up to $P$ binary places. This evaluates to 3 for IEEE single precision and 4 for both double precision and double extended formats.

### Pseudocode

The following computes the quotient of *N* and *D* with a precision of *P* binary places:

```
Express D as M × $2^e$ where 1 ≤ M < 2 (standard floating point representation)
D' := D / $2^{e+1}$   // scale between 0.5 and 1, can be performed with bit shift / exponent subtraction
N' := N / $2^{e+1}$
X := 48/17 − 32/17 × D'   // precompute constants with same precision as D
repeat $\left\lceil \log_2 \frac{P+1}{\log_2 17} \right\rceil$ times   // can be precomputed based on fixed P
    X := X + X × (1 − D' × X)
end
return N' × X
```

For example, for a double-precision floating-point division, this method uses 10 multiplies, 9 adds, and 2 shifts.

### Variant Newton–Raphson division

The Newton-Raphson division method can be modified to be slightly faster as follows. After shifting $N$ and $D$ so that $D$ is in [0.5, 1.0], initialize with

$$X := \frac{140}{33} + D \cdot \left(\frac{-64}{11} + D \cdot \frac{256}{99}\right).$$

This is the best quadratic fit to $1/D$ and gives an absolute value of the error less than or equal to 1/99. It is chosen to make the error equal to a re-scaled third order Chebyshev polynomial of the first kind. The coefficients should be pre-calculated and hard-coded.

Then in the loop, use an iteration which cubes the error.

$$\begin{aligned}&E := 1 - D \cdot X\\&Y := X \cdot E\\&X := X + Y + Y \cdot E.\end{aligned}$$

The $Y \cdot E$ term is new.

If the loop is performed until X agrees with $1/D$ on its leading $P$ bits, then the number of iterations will be no more than

$$\left\lceil \log_3\left(\frac{P+1}{\log_2 99}\right)\right\rceil$$

which is the number of times 99 must be cubed to get to $2^{P+1}$. Then

$$Q := N \cdot X$$

is the quotient to $P$ bits.

Using higher degree polynomials in either the initialization or the iteration results in a degradation of performance because the extra multiplications required would be better spent on doing more iterations.

## Goldschmidt division

Goldschmidt division[12] (after Robert Elliott Goldschmidt)[13] uses an iterative process of repeatedly multiplying both the dividend and divisor by a common factor $F_i$, chosen such that the divisor converges to 1. This causes the dividend to converge to the sought quotient $Q$:

$$Q = \frac{N}{D}\frac{F_1}{F_1}\frac{F_2}{F_2}\frac{F_{\ldots}}{F_{\ldots}}.$$

The steps for Goldschmidt division are:

1. Generate an estimate for the multiplication factor $F_i$.
2. Multiply the dividend and divisor by $F_i$.
3. If the divisor is sufficiently close to 1, return the dividend, otherwise, loop to step 1.

Assuming $N/D$ has been scaled so that $0 < D < 1$, each $F_i$ is based on $D$:

$$F_{i+1} = 2 - D_i.$$

Multiplying the dividend and divisor by the factor yields:

$$\frac{N_{i+1}}{D_{i+1}} = \frac{N_i}{D_i}\frac{F_{i+1}}{F_{i+1}}.$$

After a sufficient number $k$ of iterations $Q = N_k$.

The Goldschmidt method is used in AMD Athlon CPUs and later models.[14][15] It is also known as Anderson Earle Goldschmidt Powers (AEGP) algorithm and is implemented by various IBM processors.[16][17] Although it converges at the same rate as a Newton–Raphson implementation, one advantage of the Goldschmidt method is that the multiplications in the numerator and in the denominator can be done in parallel.[17]

### Binomial theorem

The Goldschmidt method can be used with factors that allow simplifications by the binomial theorem. Assume $N/D$ has been scaled by a power of two such that $D \in \left(\tfrac{1}{2}, 1\right]$. We choose $D = 1 - x$ and $F_i = 1 + x^{2^i}$. This yields

$$\frac{N}{1-x} = \frac{N \cdot (1+x)}{1-x^2} = \frac{N \cdot (1+x) \cdot (1+x^2)}{1-x^4} = \cdots = Q' = \frac{N' = N \cdot (1+x) \cdot (1+x^2) \cdots (1+x^{2^{(n-1)}})}{D' = 1 - x^{2^n} \approx 1}$$

.

After $n$ steps $\left(x \in \left[0, \tfrac{1}{2}\right)\right)$, the denominator $1 - x^{2^n}$ can be rounded to 1 with a relative error

$$\varepsilon_n = \frac{Q' - N'}{Q'} = x^{2^n}$$

which is maximum at $2^{-2^n}$ when $x = \tfrac{1}{2}$, thus providing a minimum precision of $2^n$ binary digits.

## Large-integer methods

Methods designed for hardware implementation generally do not scale to integers with thousands or millions of decimal digits; these frequently occur, for example, in modular reductions in cryptography. For these large integers, more efficient division algorithms transform the problem to use a small number of multiplications, which can then be done using an asymptotically efficient multiplication algorithm such as the Karatsuba algorithm, Toom–Cook multiplication or the Schönhage–Strassen algorithm. The result is that the computational complexity of the division is of the same order (up to a multiplicative constant) as that of the multiplication. Examples include reduction to multiplication by Newton's method as described above,[18] as well as the slightly faster Burnikel-Ziegler division,[19] Barrett reduction and Montgomery reduction algorithms.[20] Newton's method is particularly efficient in scenarios where one must divide by the same divisor many times, since after the initial Newton inversion only one (truncated) multiplication is needed for each division.

## Division by a constant

The division by a constant $D$ is equivalent to the multiplication by its reciprocal. Since the denominator is constant, so is its reciprocal (1/$D$). Thus it is possible to compute the value of (1/$D$) once at compile time, and at run time perform the multiplication $N$·(1/$D$) rather than the division $N/D$. In floating-point arithmetic the use of (1/$D$) presents little problem,[a] but in integer arithmetic the reciprocal will always evaluate to zero (assuming |$D$| > 1).

It is not necessary to use specifically (1/$D$); any value ($X/Y$) that reduces to (1/$D$) may be used. For example, for division by 3, the factors 1/3, 2/6, 3/9, or 194/582 could be used. Consequently, if $Y$ were a power of two the division step would reduce to a fast right bit shift. The effect of calculating $N/D$ as ($N$·$X$)/$Y$ replaces a division with a multiply and a shift. Note that the parentheses are important, as $N$·($X/Y$) will evaluate to zero.

However, unless $D$ itself is a power of two, there is no $X$ and $Y$ that satisfies the conditions above. Fortunately, $(N·X)/Y$ gives exactly the same result as $N/D$ in integer arithmetic even when $(X/Y)$ is not exactly equal to $1/D$, but "close enough" that the error introduced by the approximation is in the bits that are discarded by the shift operation.[21][22][23] Barrett reduction uses powers of 2 for the value of $Y$ to make division by $Y$ a simple right shift.[b]

As a concrete fixed-point arithmetic example, for 32-bit unsigned integers, division by 3 can be replaced with a multiply by $\frac{2863311531}{2^{33}}$, a multiplication by 2863311531 (hexadecimal 0xAAAAAAAB) followed by a 33 right bit shift. The value of 2863311531 is calculated as $\frac{2^{33}}{3}$, then rounded up. Likewise, division by 10 can be expressed as a multiplication by 3435973837 (0xCCCCCCCD) followed by division by $2^{35}$ (or 35 right bit shift).[25]: p230-234 OEIS provides sequences of the constants for multiplication as A346495 and for the right shift as A346496.

For general $x$-bit unsigned integer division where the divisor $D$ is not a power of 2, the following identity converts the division into two $x$-bit addition/subtraction, one $x$-bit by $x$-bit multiplication (where only the upper half of the result is used) and several shifts, after precomputing $k = x + \lceil \log_2 D \rceil$ and $a = \left\lceil \frac{2^k}{D} \right\rceil - 2^x$:

$$\left\lfloor \frac{N}{D} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{N-b}{2} \right\rfloor + b}{2^{k-x-1}} \right\rfloor \text{ where } b = \left\lfloor \frac{Na}{2^x} \right\rfloor$$

In some cases, division by a constant can be accomplished in even less time by converting the "multiply by a constant" into a series of shifts and adds or subtracts.[26] Of particular interest is division by 10, for which the exact quotient is obtained, with remainder if required.[27]

## Rounding error

Round-off error can be introduced by division operations due to limited precision.

## See also

- Galley division
- Multiplication algorithm
- Pentium FDIV bug

## Notes

a. Despite how "little" problem the optimization causes, this reciprocal optimization is still usually hidden behind a "fast math" flag in modern compilers as it is inexact.
b. Modern compilers commonly perform this integer multiply-and-shift optimization; for a constant only known at run-time, however, the program must implement the optimization itself.[24]

## References

1. Rodeheffer, Thomas L. (2008-08-26). *Software Integer Division* (https://www.microsoft.com/en-us/research/wp-content/uploads/2008/08/tr-2008-141.pdf) (PDF) (Technical report). Microsoft Research, Silicon Valley.
2. Morris, James E.; Iniewski, Krzysztof (2017-11-22). *Nanoelectronic Device Applications Handbook* (https://books.google.com/books?id=wAhEDwAAQBAJ&q=restoring+division+fixed-point+fractional+numbers&pg=PA243). CRC Press. ISBN 978-1-351-83197-0.
3. Shaw, Robert F. (1950). "Arithmetic Operations in a Binary Computer" (http://aip.scitation.org/doi/10.1063/1.174569

3. Shaw, Robert F. (1950). "Arithmetic Operations in a Binary Computer" (http://aip.scitation.org/doi/10.1063/1.174569 2). *Review of Scientific Instruments*. **21** (8): 690. Bibcode:1950RScI...21..687S (https://ui.adsabs.harvard.edu/abs/1950RScI...21..687S). doi:10.1063/1.1745692 (https://doi.org/10.1063%2F1.1745692). ISSN 0034-6748 (https://www.worldcat.org/issn/0034-6748). Archived (https://web.archive.org/web/20220228182241/https://aip.scitation.org/doi/10.1063/1.1745692) from the original on 2022-02-28. Retrieved 2022-02-28.
4. Flynn. "Stanford EE486 (Advanced Computer Arithmetic Division) – Chapter 5 Handout (Division)" (https://web.stanford.edu/class/ee486/doc/chap5.pdf) (PDF). *Stanford University*. Archived (https://web.archive.org/web/20220418044630/http://web.stanford.edu/class/ee486/doc/chap5.pdf) (PDF) from the original on 2022-04-18. Retrieved 2019-06-24.
5. Harris, David L.; Oberman, Stuart F.; Horowitz, Mark A. (9 September 1998). *SRT Division: Architectures, Models, and Implementations* (http://pages.hmc.edu/harris/research/srtlong.pdf) (PDF) (Technical report). Stanford University. Archived (https://web.archive.org/web/20161224030439/http://pages.hmc.edu/harris/research/srtlong.pdf) (PDF) from the original on 24 December 2016. Retrieved 23 December 2016.
6. McCann, Mark; Pippenger, Nicholas (2005). "SRT Division Algorithms as Dynamical Systems" (https://ieeexplore.ieee.org/document/614875). *SIAM Journal on Computing*. **34** (6): 1279–1301. CiteSeerX 10.1.1.72.6993 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.72.6993). doi:10.1137/S009753970444106X (https://doi.org/10.1137%2FS009753970444106X). Archived (https://web.archive.org/web/20220824213238/https://ieeexplore.ieee.org/document/614875) from the original on 2022-08-24. Retrieved 2022-08-24.
7. Cocke, John; Sweeney, D.W. (11 February 1957), *High speed arithmetic in a parallel device* (https://www.computerhistory.org/collections/catalog/102632302) (Company Memo), IBM, p. 20, archived (https://web.archive.org/web/20220824212341/https://www.computerhistory.org/collections/catalog/102632302) from the original on 24 August 2022, retrieved 24 August 2022
8. Robertson, James (1958-09-01). "A New Class of Digital Division Methods" (https://ieeexplore.ieee.org/document/5222579). *IRE Transactions on Electronic Computers*. **EC-7** (3). IEEE: 218–222. doi:10.1109/TEC.1958.5222579 (https://doi.org/10.1109%2FTEC.1958.5222579). hdl:2027/uiuo.ark:/13960/t0gt7529c (https://hdl.handle.net/2027%2Fuiuo.ark%3A%2F13960%2Ft0gt7529c). Archived (https://web.archive.org/web/20220824213239/https://ieeexplore.ieee.org/document/5222579) from the original on 2022-08-24. Retrieved 2022-08-24.
9. Tocher, K.D. (1958-01-01). "Techniques of Multiplication and Division for Automatic Binary Computers" (https://academic.oup.com/qjmam/article-abstract/11/3/364/1883426). *The Quarterly Journal of Mechanics and Applied Mathematics*. **11** (3): 364–384. doi:10.1093/qjmam/11.3.364 (https://doi.org/10.1093%2Fqjmam%2F11.3.364). Archived (https://web.archive.org/web/20220824214400/https://academic.oup.com/qjmam/article-abstract/11/3/364/1883426) from the original on 2022-08-24. Retrieved 2022-08-24.
10. "Statistical Analysis of Floating Point Flaw" (http://www.intel.com/support/processors/pentium/sb/cs-012997.htm). Intel Corporation. 1994. Archived (https://web.archive.org/web/20131023060231/http://www.intel.com/support/processors/pentium/sb/cs-012997.htm) from the original on 23 October 2013. Retrieved 22 October 2013.
11. Oberman, Stuart F.; Flynn, Michael J. (July 1995). *An Analysis of Division Algorithms and Implementations* (http://i.stanford.edu/pub/cstr/reports/csl/tr/95/675/CSL-TR-95-675.pdf) (PDF) (Technical report). Stanford University. CSL-TR-95-675. Archived (https://web.archive.org/web/20170517133304/http://i.stanford.edu/pub/cstr/reports/csl/tr/95/675/CSL-TR-95-675.pdf) (PDF) from the original on 2017-05-17. Retrieved 2016-12-23.
12. Goldschmidt, Robert E. (1964). *Applications of Division by Convergence* (http://dspace.mit.edu/bitstream/handle/1721.1/11113/34136725-MIT.pdf) (PDF) (Thesis). M.Sc. dissertation. M.I.T. OCLC 34136725 (https://www.worldcat.org/oclc/34136725). Archived (https://web.archive.org/web/20151210223340/http://dspace.mit.edu/bitstream/handle/1721.1/11113/34136725-MIT.pdf) (PDF) from the original on 2015-12-10. Retrieved 2015-09-15.
13. "Authors" (https://web.archive.org/web/20180718114413/https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5392026). *IBM Journal of Research and Development*. **11**: 125–127. 1967. doi:10.1147/rd.111.0125 (https://doi.org/10.1147%2Frd.111.0125). Archived from the original (https://ieeexplore.ieee.org/document/5392026) on 18 July 2018.
14. Oberman, Stuart F. (1999). "Floating point division and square root algorithms and implementation in the AMD-K7 Microprocessor" (http://www.acsel-lab.com/arithmetic/arith14/papers/ARITH14_Oberman.pdf) (PDF). *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No.99CB36336)*. pp. 106–115. doi:10.1109/ARITH.1999.762835 (https://doi.org/10.1109%2FARITH.1999.762835). ISBN 0-7695-0116-8. S2CID 12793819 (https://api.semanticscholar.org/CorpusID:12793819). Archived (https://web.archive.org/web/20151129095846/http://www.acsel-lab.com/arithmetic/arith14/papers/ARITH14_Oberman.pdf) (PDF) from the original on 2015-11-29. Retrieved 2015-09-15.
15. Soderquist, Peter; Leeser, Miriam (July–August 1997). "Division and Square Root: Choosing the Right Implementation" (https://www.researchgate.net/publication/2511700). *IEEE Micro*. **17** (4): 56–66. doi:10.1109/40.612224 (https://doi.org/10.1109%2F40.612224).
16. S. F. Anderson, J. G. Earle, R. E. Goldschmidt, D. M. Powers. *The IBM 360/370 model 91: floating-point execution unit*, IBM Journal of Research and Development, January 1997
17. Guy, Even; Peter, Siedel; Ferguson, Warren (1 February 2005). "A parametric error analysis of Goldschmidt's division algorithm" (https://doi.org/10.1016%2Fj.jcss.2004.08.004). *Journal of Computer and System Sciences*. **70** (1): 118

algorithm" (https://doi.org/10.1016%2Fj.jcss.2004.08.004). *Journal of Computer and System Sciences*. **70** (1): 118–139. doi:10.1016/j.jcss.2004.08.004 (https://doi.org/10.1016%2Fj.jcss.2004.08.004).

18. Hasselström, Karl (2003). *Fast Division of Large Integers: A Comparison of Algorithms* (https://web.archive.org/web/20170708221722/https://static1.squarespace.com/static/5692a9ad7086d724272eb00a/t/5692dbe6b204d50df79e577f/1452465127528/masters-thesis.pdf) (PDF) (M.Sc. in Computer Science thesis). Royal Institute of Technology. Archived from the original (https://treskal.com/s/masters-thesis.pdf) (PDF) on 8 July 2017. Retrieved 2017-07-08.
19. Joachim Ziegler, Christoph Burnikel (1998), *Fast Recursive Division* (https://domino.mpi-inf.mpg.de/internet/reports.nsf/efc044f1568a0058c125642e0064c817/a8cfefdd1ac031bbc125669b00493127/$FILE/MPI-I-98-1-022.ps), Max-Planck-Institut für Informatik, archived (https://web.archive.org/web/20110426221250/http://domino.mpi-inf.mpg.de/internet/reports.nsf/efc044f1568a0058c125642e0064c817/a8cfefdd1ac031bbc125669b00493127/$FILE/MPI-I-98-1-022.ps) from the original on 2011-04-26, retrieved 2021-09-10
20. Barrett, Paul (1987). "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor" (http://portal.acm.org/citation.cfm?id=36688). *Proceedings on Advances in cryptology---CRYPTO '86*. London, UK: Springer-Verlag. pp. 311–323. ISBN 0-387-18047-8.
21. Granlund, Torbjörn; Montgomery, Peter L. (June 1994). "Division by Invariant Integers using Multiplication" (http://gmplib.org/~tege/divcnst-pldi94.pdf) (PDF). *SIGPLAN Notices*. **29** (6): 61–72. CiteSeerX 10.1.1.1.2556 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.2556). doi:10.1145/773473.178249 (https://doi.org/10.1145%2F773473.178249). Archived (https://web.archive.org/web/20190606211506/https://gmplib.org/~tege/divcnst-pldi94.pdf) (PDF) from the original on 2019-06-06. Retrieved 2015-12-08.
22. Möller, Niels; Granlund, Torbjörn (February 2011). "Improved Division by Invariant Integers" (http://gmplib.org/~tege/division-paper.pdf) (PDF). *IEEE Transactions on Computers*. **60** (2): 165–175. doi:10.1109/TC.2010.143 (https://doi.org/10.1109%2FTC.2010.143). S2CID 13347152 (https://api.semanticscholar.org/CorpusID:13347152). Archived (https://web.archive.org/web/20151222160554/https://gmplib.org/~tege/division-paper.pdf) (PDF) from the original on 2015-12-22. Retrieved 2015-12-08.
23. ridiculous_fish. "Labor of Division (Episode III): Faster Unsigned Division by Constants" (http://ridiculousfish.com/files/faster_unsigned_division_by_constants.pdf) Archived (https://web.archive.org/web/20220108225258/http://ridiculousfish.com/files/faster_unsigned_division_by_constants.pdf) 2022-01-08 at the Wayback Machine. 2011.
24. ridiculous_fish. "libdivide, optimized integer division" (https://libdivide.com/). Archived (https://web.archive.org/web/20211123015446/https://libdivide.com/) from the original on 23 November 2021. Retrieved 6 July 2021.
25. Warren Jr., Henry S. (2013). *Hacker's Delight* (2 ed.). Addison Wesley - Pearson Education, Inc. ISBN 978-0-321-84268-8.
26. LaBudde, Robert A.; Golovchenko, Nikolai; Newton, James; and Parker, David; *Massmind: "Binary Division by a Constant"* (http://techref.massmind.org/techref/method/math/divconst.htm) Archived (https://web.archive.org/web/20220109215748/http://techref.massmind.org/techref/method/math/divconst.htm) 2022-01-09 at the Wayback Machine
27. Vowels, R. A. (1992). "Division by 10". *Australian Computer Journal*. **24** (3): 81–85.

## Further reading

- Savard, John J. G. (2018) [2006]. "Advanced Arithmetic Techniques" (http://www.quadibloc.com/comp/cp0202.htm). *quadibloc*. Archived (https://web.archive.org/web/20180703001722/http://www.quadibloc.com/comp/cp0202.htm) from the original on 2018-07-03. Retrieved 2018-07-16.