# Constant-division algorithms

P. Srinivasan
F.E. Petry

*Indexing terms: Address mapping, Base conversion, Fast-division techniques, Normalisation*

**Abstract:** There exist many types of special-purpose systems that require rapid and repeated division by a set of known constant divisors. Numerous solutions have been proposed in response to the deficiencies of the conventional division algorithms for applications which involve repeated divisions by known constants. Six approaches are reviewed in detail and their relationships are shown by reducing them to equivalent forms. Proving the equivalence of these algorithms allows them to be considered as alternative implementations of the same basic function. Proof of correctness of one form serves to verify all the methods. The analytical process has led to an improved understanding of constant division and of the division operation in general. It has provided a foundation for further analysis and algorithm development, including the establishment of the theoretical basis of quotient and remainder generation, a generalised implementation of division by divisors $2^n \pm 1$, and extension of this method to divide by small integers by generating the value of the $B$-sequence, the value in one period, of the integer reciprocal.

## 1 Introduction

Many types of special-purpose systems require rapid and repeated division by a set of known constant divisors. These include scaling for normalisation and exponent alignment in nonbinary bases [1–4], conversion between bases [5, 6], address mapping in bit-addressable, or interleaved, memory schemes (usually with a prime number of modules) [7, 8], element location in multidimensional lists, and the solution of mathematical equations in many algorithms, for example those that process telephone traffic [9]. Even in general-purpose machines, faster execution of specific division cases improves the average performance of division [10]. Some of the complexity characteristics of integer division by constants have been considered in the framework of an abstract computational model [11]. Numerous solutions have been proposed in response to the deficiencies of the conventional division algorithms for applications which involve repeated divisions by known constants.



Paper 1414E (C1, C8, C14), first received 17th September 1993 and in revised form 24th May 1994

P. Srinivasan is with the Computer Science Department, University of Southwestern Louisiana, Lafayette, LA 70504-1771, USA

F.E. Petry is with the Computer Science Department, Tulane University, New Orleans, LA 70118, USA

The application-dependent development and ad hoc presentation of most constant-division approaches has left a gap in the theoretical understanding of the more general process, so six approaches are reviewed in this paper and reduced to equivalent forms. Demonstrating the equivalence of these algorithms allows alternate implementations of the same basic function to be considered and a proof of correctness of one form then serves to verify all of the methods. This analytical process has led to an improved understanding of constant division and of the division operation in general, and has provided a foundation for further analysis and algorithm development [12, 13].

A majority of the constant-division algorithms developed have dealt with single divisor values or with sets of 'well-formed' divisors, for reasons derived from the motivating application or owing to computational advantages specific to the form considered. The six methods considered vary appreciably in the range of acceptable inputs, the manner of handling of inexact divides, remainder generation and the direction from which quotient digits are generated. Three [1, 5, 6] out of these six were constructed to divide by the specific value ten and the other three were constructed to handle an arbitrary integer [14] or divisors of the form $2^n \pm 1$ [7, 8]. Some methods carry out only a subset of the functions involved in division, for example that of determining the quotient solely for an exact divide. This information is summarised in Table 1 in which the functions shown are common to all the algorithms facilitating the analysis of the approaches in the following Sections.

The notation used to express these algorithms has been standardised as follows: $l$ = wordlength, $D$ = dividend, $d$ = divident bits, $a$ = dividend digits base $B$, $B$ = base, $\delta$ = divisor, $R$ = remainder, $Q$ = quotient, $q$ = quotient bits, and $e$ = quotient digits base $B$.

## 2 Constant-division algorithms overview

The six constant-division algorithms summarised in Table1 are reviewed in the following Sections.

| | | | | | | |
|---|---|---|---|---|---|---|
| + | $a_{m-2}$ | $a_{m-3}$ | ⋯ | $a_2$ | $a_1$ | $a_0$ |
| − | $a_{m-3}$ | $a_{m-4}$ | ⋯ | $a_1$ | $a_0$ | 0 |
| + | $a_{m-4}$ | $a_{m-5}$ | ⋯ | $a_0$ | 0 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| − | $a_1$ | $a_0$ | ⋯ | 0 | 0 | 0 |
| + | $a_0$ | 0 | ⋯ | 0 | 0 | 0 |
| | $e_{m-2}$ | $e_{m-3}$ | ⋯ | $e_2$ | $e_1$ | $e_0$ |

**Fig. 1** *Division by five in terms of dividend digits*

### 2.1 Method 1

First we discuss a technique [6] for dividing a positive binary integer by ten for decimal–binary conversion. The algorithm uses a shift followed by a division by five. The

division by five proceeds as follows: For an exact divide the two low-order bits of the quotient are equal to the two low-order bits of the dividend. These bits are subtracted from the rest of the dividend bits to produce two more quotient bits. In the case of a nonzero remainder, it must be determined and subtracted from the dividend before the quotient can be computed using this method. The remainder $R = D \bmod 5$, is derived by adding the weighted bits of the dividend.

We restate this method in terms of digits base four of the dividend $D$ by grouping its bits into pairs, giving dividend digits $a$ with base $B$ equal to four: $D = a_{m-1}a_{m-2}a_{m-3}\cdots a_1 \cdot a_0$, $m = \lceil l/2 \rceil$. The subtraction from the dividend digits of the quotient digits as they are developed from the right is expressed in Fig. 1 and represented in eqn. 1:

$$Q = \sum_{i=0}^{m-1} (-1)^i 2^{2i} D \tag{1}$$

### 2.2 Method 2

The second technique [1, 3] is part of the shifting and scaling (normalisation) mechanism in a decimal floating-point system, and also produces quotient bits from the rightmost or low-order bit. The shift-subtract 'transformation' is very similar to the computation proposed in Reference 6. The methods differ, however, in the way the remainder is handled as here the remainder need not be determined and subtracted before the transformation can be applied. Instead, the transform of the remainder, i.e. the remainder value subjected to the shift-subtract operation described, is subtracted from the transformed dividend. The correct remainder value transform is determined by generating high-order bits in the shift subtract process beyond the largest possible quotient value. The shift-subtract process applied to an integer remainder produces a repeating pattern of $n$ bits width. The pattern in these high-order bits is therefore sufficient for the remainder value to be clearly determined for subtraction and decoding into an integer remainder.

This method has been shown to be extensible to divisors of the form $2^n \pm 1$ [12], expressed as $2^n - h$, fixed $h = \pm 1$. In an exact divide, the two methods can be treated identically at an algorithmic level. The resulting more general summation may be expressed as

$$Q = -h \sum_{i=0}^{m-1} (h)^i 2^{in} D \tag{2}$$

where $h = \pm 1$, the divisor $\delta = 2^n - h$, and $m = \lceil l/n \rceil$.

### 2.3 Method 3

Another method in which the quotient is determined from the low-order bits is demonstrated as a software or microcode routine for divisors of the form $2^n \pm 1$ [7].

**Table 1: Comparison of division algorithms**

| Ref. | (i) | (ii) | (iii) | (iv) |
|---|---|---|---|---|
| 2.1 [6] | Ten | Exact divides only | Not computed | Low to high |
| 2.2 [1, 3] | Ten, $2^n \pm 1$* | Single subtract of transformed remainder | Must decode transformed remainder | Low to high |
| 2.3 [7] | $2^n \pm 1$ | Maximum of $d$ subtractions required | Count of actual subtractions | Low to high |
| 2.4 [14] | Positive $2^n - 1$† | No additional steps | $R/d$ computed, multiply by $d$ | High to low |
| 2.5 [5] | Ten | No additional steps | $R$ computed directly | High to low |
| 2.6 [8] | $2^n \pm 1$ | No additional steps | $R$ computed directly | Convergence/high to low |

* Extended in a related research effort
† *Actual algorithm steps shown for this form only*
(i) Divisor domain
(ii) Quotient determination for nonzero remainders
(iii) Remainder determination
(iv) Order of quotient digit generation (high/low-order digits first)

The method was developed for an address-mapping application that required the treatment of an inexact divide as an error condition. While it easily determines the value of the quotient in an exact divide and the existence of a nonzero remainder, this method does not easily detect the *value* of the remainder or that of the quotient in an inexact divide. In such an inexact divide situation, the quotient and remainder can both be found by at most $d$ comparisons and subtractions.

Quotients less than $2^n$ are determined directly from the low-order bits of the dividend. Dividends with length greater than $2n$ are multiplied by $2^n + 1$ or $2^n - 1$, for divisor values $2^n - 1$ and $2^n + 1$, respectively, and the two's complement of the result modulo $2^{2n}$ is the quotient. The initial multiplication serves to standardise the two divisor values to $2^{2n} - 1$. The two divisors can be dealt with uniformly from this point. To process larger dividend values the product from the previous stage must be multiplied by $2^{2n} + 1$, $2^{4n} + 1$, $2^{8n} + 1$, etc., doubling the maximum dividend width for which the quotient is valid in each step. This may be viewed as developing quotient digits from the right.

The algorithm for division of a dividend $D$ by integers of the form $2^n - h$, fixed $h = \pm 1$, is restated as the following computation for $m = \lceil l/n \rceil$, $k = \lceil (\log_2 m) - 1 \rceil$:

$$R_0 = D * (2^n + h)$$

$$R_1 = R_0 * (2^{2n} + 1)$$

$$\cdots$$

$$R_k = R_{k-1}(2^{2^{k-1}n} + 1)$$

and the quotient: $Q =$ two's complement $(R_m)$, can be expressed as

$$Q = 2^l - D(2^n - y)(2^{2n} + 1)(2^{4n} + 1)\ldots(2^{2^{k-1}n} + 1)$$

or

$$Q = -(2^n + h)D\left(\prod_{j=1}^{k} 2^{2^j n} + 1\right) \tag{3}$$

### 2.4 Method 4

A more generally motivated combinational algorithm has been developed for division by fixed integers using multiplication by a fractional inverse [14]. The reciprocal of the odd divisor $\delta$ is recognised to be a repeating fraction,

with cycle length or period $n$. The dividend is multiplied by one period of the reciprocal in the standard manner and by successive periods using a series of shifts and adds. No consideration has been given to the manner of generation of this inverse. What is presented therefore is a method of multiplying the dividend by the successive periods of a given inverse, after one period of the inverse has been multiplied into the dividend, using $n$-bit digit-serial shift and add operations, where $n$ is the period of the reciprocal. Carries are handled in a separate step. We have studied a method of inverse generation, as a useful extension of the knowledge surveyed.

For an adjusted dividend $D' = a'_{m-1}a'_{m-2}a'_{m-3}\cdots a'_1 \cdot a'_0$ consisting of $m$ $n$-bit digits, where $m = \lceil l/n \rceil$ and $n$ is the period of the reciprocal; the computation is illustrated in Fig. 2. The summation that is seen in the Figure may be expressed as

$$D'\sum_{i=1}^{\infty} 2^{-in} = DS'\sum_{i=1}^{\infty} 2^{-in}$$

$$\begin{array}{llllll}
 & a'_{m-1} & a'_{m-2}a'_{m-3} & \cdots & a'_1 & \cdot\ a'_0 \\
+ & & a'_{m-1}a'_{m-2} & \cdots & a'_2 & \cdot\ a'_1a'_0 \\
+ & & & a'_{m-1} & \cdots\ a'_3 & \cdot\ a'_2a'_1a'_0 \\
+ & & & & a'_{m-1} & \cdot\ a'_{m-2}\ \cdots\ a'_2a'_1a'_0 \\
+ & & & & & \cdot\ a'_{m-1}\ \cdots\ a'_3a'_2a'_1a'_0 \\
\hline
 & e_{m-1} & e_{m-2} & e_{m-3} & \cdots\ e_0 & \cdot\ R/\delta
\end{array}$$

**Fig. 2** *Multiplication by successive periods of an infinitely repeating reciprocal*

The quotient portion of this result is the floor of the preceding expression:

$$Q = \left\lfloor DS'\sum_{i=1}^{\infty} 2^{-in} \right\rfloor \tag{4}$$

### 2.5 Method 5

Another approach provided the design for a binary–decimal converter incorporating a fast algorithm for division by the specific value *five* [5]. In this approach the quotient bits are developed from the left with a serial 'modified subtraction', using the quotient bit or digit developed at one stage as the input into the next stage.

In the divide by five from the left presented, dividend and quotient bits are considered to be grouped by two to form digits base four, with bounds zero and three. The modified subtraction is applied with dividend digits $a_i$, quotient digits $e_i$, borrow bits $c$, and a temporary value $s_i$, in $s_i = a_i - e_i + 4c_i = e_{i-1} + c_{i-1}$. The values of $s_i$ are bounded by zero and four. Within these bounds are multiple combinations of values which satisfy the equation. These ambiguous situations are resolved by looking ahead one position and imposing that $s_{i-1} = a_{i-1} - e_{i-1} + 4c_{i-1}$ stays within the same limits of zero and four. The result is then the sum of the shifted quotient and the remainder and the least significant digit contains the remainder. The remainder may take values between 0 and $2^n$ and the remainder of $2^n$ results when $e_0 = a_0 + 1$.

Although this approach implies digit-serial processing, the computation can be expressed solely in terms of dividend digits, base 4. The high- to low-order generation of digits, from $m = \lceil l/2 \rceil$, is illustrated in Fig. 3 and the summation shown there can be summarised in eqn. 5:

$$Q = \left\lfloor \sum_{i=1}^{m} (-1)^{i-1}2^{-2i}D \right\rfloor \tag{5}$$

### 2.6 Method 6

An iterative algorithm [8] has been developed for division by numbers of the forms $2^n \pm 1$, represented as $2^n - h$, where $h = \pm 1$. The dividend bits are grouped and

$$\begin{array}{llllll}
+ & a_{m-1} & a_{m-2} & a_{m-3} & \cdots\ a_1 & \cdot\ a_0 \\
- & & a_{m-1} & a_{m-2} & \cdots\ a_2 & \cdot\ a_1 \\
\vdots & & & & \cdots\ \vdots & \vdots \\
- & & & & a_{m-1} & \cdot\ a_{m-2} \\
+ & & & & & \cdot\ a_{m-1} \\
\hline
 & e_{m-2} & e_{m-3} & e_{m-4} & \cdots\ e_0 & \cdot\ R
\end{array}$$

**Fig. 3** *High- to low-order generation division in terms of dividend digits*

partial quotient and partial remainder are produced in independent steps, using shifts and add/subtracts according to the following formulas:

$$Q^{(0)} = 0$$
$$R^{(0)} = A_1^{(0)}A_2^{(0)}\cdots A_k^{(0)} \quad k_0 = \lceil L^{(0)}/n \rceil$$

$$Q^{(j+1)} = Q^{(j)} + \sum_{i=1}^{k_j-1} h^{i-1}A_1^{(j)}A_2^{(j)}\cdots A_{ji}^{(j)} \tag{6}$$

$$R^{(j+1)} = \sum_{i=1}^{k_j-1} h^{i-1}A_{kj-i+1}^{(j)} = A_1^{(j+1)}A_2^{(j+1)}\cdots A_{kj+1}^{(j+1)}$$
$$k_{j+1} = \lceil L^{(j+1)}/n \rceil \quad j = 0, 1, 2, \ldots \tag{7}$$

where $Q$ is the quotient, $R$ the partial remainder, $L$ the length of the partial remainder, and $k$ the number of $n$-bit digits. Successive iterations use the partial remainders as input, and the algorithm halts when the partial remainder consists of a single bit group or digit. The stopping condition of partial remainder being $n$ bits or less in length may not ensure that the remainder value remains within the allowed range of a remainder for a divisor $2^n - 1$. A correction is therefore needed at the end for such situations.

The computation of the partial remainder in the second iterative step (eqn. 7), expands as follows:

$$a_0 + h^1 * a_1 + h^2 * a_2 + \cdots + h^{n-2} * a_{m-2} + h^{n-1} * a_{m-1} \tag{8}$$

The powers of $h$ in the terms make the series an alternating series for $h = -1$, or $\delta = 2^n + 1$. The result of the remainder step is subjected to a series of shift adds as illustrated and added to the previous quotient estimate. Putting this computation together with the quotient step (eqn. 6) and allowing the carry from the partial remainder computation to propagate over into the quotient portion with endaround carries/borrows where necessary will result in the single combined implementation in Fig. 4.

$$\begin{array}{llllll}
 & a_{m-1} & a_{m-2} & a_{m-3} & \cdots\ a_1 & \cdot\ a_0 \\
+\ h * & & a_{m-1} & a_{m-2} & \cdots\ a_2 & \cdot\ a_1 \\
+\ h^2 * & & & a_{m-1} & \cdots\ a_3 & \cdot\ a_2 \\
\vdots & & & & \cdots\ \vdots & \vdots \\
+\ h^{m-2} * & & & & a_{m-1} & \cdot\ a_{m-2} \\
+\ h^{m-1} * & & & & & \cdot\ a_{m-1} \\
\hline
 & e_{m-2} & e_{m-3} & e_{m-4} & \cdots\ e_0 & \cdot\ R
\end{array}$$

**Fig. 4** *Iterative division by $2^n \pm 1$*

The computation of the quotient may be summarised as

$$Q = \left\lfloor \sum_{i=1}^{m} (h)^{i-1}2^{-in}D \right\rfloor \tag{9}$$

## 3 Unification of division approaches

Now we show the equivalence of the statements of these algorithms for generating the quotient in an exact divide

by developing a common form for the first three methods, unifying the last three, and then showing equivalence of the two resulting forms.

### 3.1 Unification of methods 1–3

The divisor value addressed in eqn. 1, i.e. five, is clearly covered in eqn. 2, and corresponds to the divisor $2^n - h$, $h = -1$, $n = 2$. The statement of the method in eqn. 3 contains as its first portion the term $(2^n - h)D$, for $\delta = (2^n + h)Q$, resulting in $(2^n - 1)(2^n + 1)Q$, no matter which value $h$ takes in the divisor. Therefore

$$Q = (2^n - h)D\left(\prod_{j=1}^{k} 2^{2jn} + 1\right)$$
$$= -(2^n - 1)(2^n + 1)Q\left(\prod_{j=1}^{k} 2^{2jn} + 1\right)$$
$$= -(2^n - 1)Q\left(\prod_{j=0}^{k} 2^{2jn} + 1\right)$$

This product form is shown to be equivalent to the additive series form for division by $2^n \pm 1$ presented previously in eqn. 2, recalling that $k = \lceil \log_2 l/n - 1 \rceil$, $m = \lceil l/n \rceil$, and that mod $2^l$ operations are assumed. Since the product form presents the restated dividend as $(2^n - 1)Q$, we examine the summation for the divisor value $\delta = 2^n - 1$, or $h = 1$, given also that only exact divides are handled in this computation:

$$Q = -\sum_{i=0}^{m-1} 2^{in} D$$
$$= -(2^n - 1)Q \sum_{i=0}^{m} 2^{in}$$

Inspection of the two forms considered shows that, if

$$\prod_{i=0}^{k} (2^{2^{in}} + 1) \equiv \sum_{i=0}^{m} 2^{in}$$

the expressions (and therefore the algorithms) may be said to be equivalent. The two forms are easily proven equivalent using $k = \lceil \log_2(l/n) \rceil$, and therefore $2^k < m \leqslant 2^{k+1}$, and $l \leqslant 2^{k+1} n$. The finite product expands into a series with terms identical to the sum.

### 3.2 Unification of methods 4–6

Now consider for unification the last three methods, in which quotient digits are generated in a high-to-low order. The inverse of every odd integer $\delta$ greater than one is a repeating binary fraction $S$ of some period $n$ bits such that the value in one period is the minimum integer $S'$ such that $\delta S' = 2^n - 1$ [7]. The computation considered in Reference 14 may be restated using this result:

$$Q = \left\lfloor DS' \sum_{i=1}^{\infty} 2^{-in} \right\rfloor$$
$$= \left\lfloor Q\delta S' \sum_{i=1}^{\infty} 2^{-in} \right\rfloor = \left\lfloor Q\delta' \sum_{i=1}^{\infty} 2^{-in} \right\rfloor = \left\lfloor D' \sum_{i=1}^{\infty} 2^{-in} \right\rfloor$$

where $\delta' = \delta S' = 2^n - 1$, and $D' = D \cdot S'$. The multiplication by $S'$ may be seen as converting the division of $D$ by $\delta$ to a division of $D'$ by $2^n - 1$, where $n$ is the period of the reciprocal of $\delta$. This statement of the computation of the quotient is an extension to infinity of the summation for the method presented [8] for division by $2^n - h$, $h = 1$, as shown in eqn. 9. The expansion of the infinite sum beyond the $m$th term in eqn. 9 can be seen to produce a value that does not affect the floor or integer portion of the result, i.e.

$$\left\lfloor D \sum_{i=1}^{m} 2^{-in} \right\rfloor = \left\lfloor D' \sum_{i=1}^{\infty} 2^{-in} \right\rfloor$$

since

$$\left\lfloor D' \sum_{i=1}^{\infty} 2^{-in} \right\rfloor = \left\lfloor D' \sum_{i=1}^{m} 2^{-in} + D' \sum_{i=m+1}^{\infty} 2^{-in} \right\rfloor$$

and

$$\left\lfloor D' \sum_{i=m+1}^{\infty} 2^{-in} \right\rfloor = 0$$

given $D < 2^l \leqslant 2^{mn}$. This is shown formally in Appendix 8.1.

Note that a method to compute the value of $S'$, or the value in one period of the reciprocal, is not provided and has been addressed in Section 5. What we essentially have therefore is a $2^n - 1$ divider design. The expression to compute the quotient in a division by five in the algorithm in Reference 5 is also clearly covered in the statement of the computation in Reference 8, for divisors $2^n - h$, $h = -1$, $n = 2$.

### 3.3 General unification

The following unified forms have been presented for algorithms that develop quotient digits from the low- and high-order digits (right and left divides), respectively:

$$Q_{rt} = -h \sum_{i=0}^{m-1} h^i 2^{in} D \quad Q_{lf} = \left\lfloor \sum_{i=1}^{m} h^{i-1} 2^{-in} D \right\rfloor$$

We now relate these generalised left and right computations. First, examine the expansions of these computations as applied to an example dividend consisting of five base $2^n$ digits (named A, B, C, D, and E for convenience). The two computations are presented side by side in Fig. 5 for comparison.

Discounting the carry propagation, the computation resulting in the $i$th quotient digit in each of the two general approaches are shown in eqns. 10 and 11 for $D = 2^0 x_0 + 2^n x_1 + \cdots + 2^{(m-1)n} x_{m-1}$.

$$t_i = -\sum_{j=0}^{k-1} h^i x_j \pmod{2^n - h} \tag{10}$$

$$u_i = \sum_{i=k}^{m-1} h^i x_i \pmod{2^n - h} \tag{11}$$

In the exact divide situation, $D = 0 \pmod{\delta}$, or the remainder $R = 0$. One method for computing this remainder, or $D \pmod{\delta}$, denoted $|D|_\delta$, is to first calculate and store the contribution to the remainder of each bit position, and then sum (mod $\delta$) the contribution of

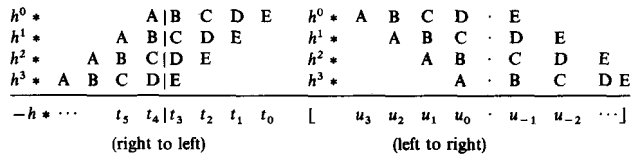


**Fig. 5** *Generalised divides from right and left*

each nonzero bit. The method more commonly used for the mod operation with moduli of the form $2^n \pm 1$ in applications such as residue number systems [15] is to sum the contribution of base $2^n$ digits instead of proceeding bit by bit. This approach takes advantage of the fact that each digit contributes its own value to the residue mod $2^n - 1$, and each contributes its value with alternating sign changes to the residue mod $2^n + 1$ ($|2^{kn}|_{2^n+1} = (-1)^k$, and $|(2^{kn})|_{2^n-1} = 1$). Finding the residue is thereby simplified to summing the digits of the dividend mod the divisor, with alternating sign changes for modulus $2^n + 1$. Therefore to find $D$ modulo the divisor $\delta$, where $\delta = 2^n - h$, $D \pmod{2^n - 1} = (h^0x_0 + h^1x_1 + h^2x_2 + \cdots + h^{m-1}x_{m-1}) \bmod 2^n - h$, or

$$\sum_{i=0}^{m-1} h^i x_i \pmod{2^n}$$

Since the remainder is known to be zero, for a dividend expressed base $2^n$ or in $n$-bit digits, $D = 2^0x_0 + 2^nx_1 + 2^{2n}x_2 \cdots + 2^{(m-1)n}x_{m-1}$ one can say

$$\sum_{i=0}^{m-1} h^i x_i \pmod{2^n - h} = 0$$

and

$$\sum_{i=k}^{m-1} h^i x_i \pmod{2^n - h} = -\sum_{j=0}^{k-1} h^i x_j \pmod{2^n - h} \qquad (12)$$

Note the correspondence between the two terms in eqn. 12 and the expressions for $t_i$ and $u_i$ in eqns. 10 and 11. However, the two are equivalent only if the computation is carried out mod $2^n - h$ over the column. The modular arithmetic is in fact simulated through the carry/borrow that comes into a digit position from the column to the right. We illustrate for $\delta = 2^n - 1$: $u_i = ((A + B + C + D) \bmod 2^n) + u_{i-1}(\text{div } 2^n)) \bmod 2^n)$.

This is equivalent in behaviour to an end-around carry, which would, for every carry generated in a particular column, increment the same column. The rightmost or least-significant bit group (into which no carry can propagate) must have an explicit end-around carry if the algorithm is implemented in this fashion.

Similarly, for $\delta = 2^n + 1$, the mod $2^n + 1$ arithmetic is implemented through the carry/borrow in from the digit to the right in the base $2^n$ hardware. This is equivalent to an end-around carry with a sign change, i.e. a carry out of a digit position results in a decrement of that digit, and vice versa.

Since the two summation are shown to be equivalent mod $2^n - h$, and the arithmetic is indeed carried out mod $2^n - h$, then one can say that the summation operations carried out in each column in both right and left divides deliver the same values for quotient digits, and therefore

$$Q = -h\sum_{i=0}^{m-1} h^i 2^{in} D \pmod{2^l} = \left\lfloor \sum_{i=1}^{m} h^{i-1} 2^{-in} D \right\rfloor$$

## 4 Reminder generation

As seen from Table 1, of the six methods surveyed, the first [6] ignores the issue entirely, the next two [1, 7] involve additional computations independent of the quotient development procedure, to retrieve the remainder as well as to compute the quotient in an inexact divide. The fourth method [14] produces, using the same basic step used in computing the quotient, the fraction $R/\delta$, which must be multiplied by $\delta$ to determine the remainder. Finally, the last two methods, [5, 8] develop an integer remainder $R$ using the same basic step as for quotient generation. We examine these last two approaches to remainder generation. For a dividend $D$ represented in base $2^n$ digits as $D = 2^0x_0 + 2^nx_1 + 2^{2n}x_2 \cdots + 2^{(m-1)n}x_m$, $m = \lceil l/n \rceil$, the remainder is computed in References 5 and 8 in the least significant column of the computation presented in Fig. 3 and 4, which corresponds to $h^0x_0 + h^1x_1 + h^2x_2 + \cdots + h^{m-1}x_{m-1}$ with an end-around carry in this digit. With normal carry propagation (leaving aside the effect of the end-around carry), this computation may be summarised as

$$\sum_{i=0}^{m-1} h^i x_i \pmod{2^n}$$

The end-around carry in the least-significant digit accomplishes the modulo $2^n \pm 1$ operation in the computation of the remainder on modulo $2^n$ hardware. This end-around carry $c$, has the value 1 when there is a carry 0, when there is no carry, and $-1$ when there is a borrow. The end-around carry is implemented with a sign change for divisor $2^n + 1$, or $h = -1$. The computation may be presented as

$$\sum_{i=0}^{m-1} h^i x_i \pmod{2^n} + c_i h = \sum_{i=0}^{m-1} h^i x_i \pmod{2^n - h}$$

where $h = \pm 1$, the divisor $\delta = 2^n - h$, and $c = \{-1, 0, 1\}$. The remainder value is generated in Reference 14 in the most significant digit of the continuing fraction representation of $R/\delta$. Generation of the rest of the digits and multiplication by the divisor is therefore not necessary.

## 5 Related results

The analysis of the various constant division algorithms surveyed has provided a good foundation for further algorithm development and the exploration of theoretical issues related to the nature of constant division. Some specific results that have arisen from this study are enumerated in this Section.

### 5.1 Validity of quotient and remainder results

Starting from the summation form of the division method in eqn. 9, a formal demonstration of the validity of the quotient and remainder computations is presented in Appendix 8.1 [16]. The expression for the algorithm is shown algebraically to be equivalent to the quotient plus some term which will not affect the floor of the expression, or the value of the quotient.

### 5.2 Implementation of division by $2^n + 1$

An important and direct outcome of this research has been the definition of improved approaches to division by constants belonging to the set of integers of the form $2^n + 1$. Based on Stefanelli's binary–decimal conversion algorithm [5], a generalised algorithm for division by integers of the form $2^n \pm 1$, for an arbitrary value or values of $n$, has been developed to produce precise integer quotients and remainders. The computation may be presented as a multiplication of the dividend $D$ by the summation discussed (which represents the inverse of the divisor). Implementation issues have been explored in the development of a VLSI-oriented design for the constant division method [13, 16].

Bit-level equations for the $2^n - 1$ divider follows. Note that $n$ bits at most must be considered to produce one quotient bit.

$$q_{i-n} = d_i \oplus q_i \oplus b_i$$

where

$$b_i = G_i \vee (P_i \wedge G_{i-1}) \vee (P_i \wedge P_{i-1} \wedge G_{i-2}) \vee \cdots \vee (P_i \wedge P_{i-1} \wedge \cdots \wedge P_{i-n+1} \wedge G'_{i-n})$$

and

$$G_i = q_i \wedge d_i \quad P_i = \bar{q}_i \vee d_i \wedge q_i \wedge \bar{d}_i \quad G'_{i-n} = d_{i-n} \vee q'_{i-n}$$

where $q'_{i-n} = d_i \oplus q_i$. These bit-level equations can be applied in an overlapped manner to carry out partial calculations for $n$ bits at a time. The designs can also be combined with a selection in computing $G_i$ and $P_i$ depending on the value of the divisor. Another alternative implementation is at the digit level using an adder tree.

### 5.3 Division by small integer variables

While the constant form $2^n \pm 1$ is important in that it covers many integers encountered frequently in computations involving constants, it is important to be able to generalise the division algorithm to accommodate other forms as well as to remove the predetermined constant restriction. Dividers designed for constant divisors in this set can trivially be extended to cover divisors of the form $2^m(2^n \pm 1)$, $m \in N$ by the incorporation of an $m$-bit shift and concatenation of the shifted out bits to the remainder result.

To extend these algorithms to incorporate arbitrary integer values as divisors, the Euler–Fermat theorem can be used to show that, with one additional multiplication, division by an arbitrary integer can be converted to a division by an integer of the form $2^n - 1$. The theorem is stated as follows: If $a$ is any integer relatively prime to the modulus $k$, then $a^{\phi(k)} \equiv 1 \pmod{k}$, where $\phi$ is the well known Euler $\phi$-function [17]. From this one can say that for every $x \in N$ there exist three positive integers $m$, $n$ and $s$ such that $xs = 2^m(2^n - 1)$, or for every odd integer divisor $y$ there exist two positive integers $n$ and $s$ such that $ys = 2^n - 1$ or $2^n \equiv 1 \pmod{y}$. One can therefore conclude that division by any integer can be converted to division by an integer of the form $2^m(2^n \pm 1)$ by means of one additional multiplication.

The inverse of every odd integer greater than one is a fraction whose binary representation is of the form $\cdots s_1 s_2 \cdots s_n s_1 s_2 \cdots s_n \cdots$. The bits $s_1 s_2 \cdots s_n$ are the representation of the minimum integer $s$ such that $ds = 2^n - 1$ [7]. The multiplier to be determined $s$ can therefore be established to be the value in one period of the reciprocal of the divisor, referred to as $B$-sequence. The problem reduces to one of $B$-sequence determination. An approach to $B$-sequence determination has therefore been developed, and the algorithm and an implementation summary follow. The availability of the $B$-sequence generation technique allows the $2^n - 1$ divider to be used with an unknown integer divisor. The method is recommended for small integer divisors because of its dependence on the length of the $B$-sequence or the period of the reciprocal.

Initial step:

$q_0 = 1$, $u = 0$, $y = \lceil d/2 \rceil + 1$, $t_0 = y$, $i = 0$

while $t_i \neq 1$ do begin

$q_i := t_i \pmod 2$

$u_i := \lceil t_i/2 \rceil$

$t_i + 1 := y * q_i + u_i = i + 1$

$i := i + 1$

end

A systolic array implementation of this reciprocal generation and division method is available. The division process therefore consists of the generation of the recurring sequence in the repeating binary reciprocal of the divisor, followed by a multiplication of the dividend $D$ by this sequence to produce $D'$, and finally a division of $D'$ by $2^n - 1$, where $n$ is the length of the recurring sequence, or the period of the reciprocal.

## 6 Conclusions

The various constant division algorithms surveyed have been shown to be equivalent by first reducing them to their mathematical forms, establishing first the equivalences among two subsets of the set of algorithms, and finally showing the equivalence of the two resulting forms. Some research developments resulting from this survey and analysis have been outlined, including the establishment of the theoretical basis of quotient and remainder generation, a generalised implementation of division by divisors $2^n \pm 1$, and extension of this method to divide by small integers by generating the value of the $B$-sequence of the integer reciprocal.

## 7 References


1 JOHANNES, J., PEGDEN, C., and PETRY, F.: 'Decimal shifting for an exact floating point representation', *Comput. Electr. Eng.*, Sept. 1980, **7**, (3), pp. 149–155

2 JOHNSTONE, P., and PETRY, F.: 'Design and analysis of nonbinary-radix floating point representations', *Comput. Electr. Eng.*, 1994, **20**, (1), pp. 39–50

3 PETRY, F.E.: 'Two's complement extension of a parallel binary division by ten', *Electron. Lett.*, September 1983, **19**, (18), pp. 718–720

4 BOHLENDER, G.: 'Decimal floating-point arithmetic in binary representation', *in* KAUCHER, E., MARKOV, S., and MAYER, G. (Eds.): 'Computer arithmetic' (Baltzer, 1991), pp. 13–27

5 SCHREIBER, F.A., and STEFANELLI, R.: 'Two methods for fast integer binary-BCD conversion'. Proceedings of the Fourth IEEE Symposium on *Computer Arithmetic*, Santa Monica, CA, 1978, pp. 200–207

6 SITES, R.: 'Serial binary division by ten', *IEEE Trans.*, 1974, **C-23**, (12), pp. 1299–1301

7 ARTZY, E., HINDS, J.A., and SAAL, H.J.: 'A fast division technique for constant divisors', *Comm. ACM*, February 1976, **19**, (2), pp. 98–101

8 CI YUN GUEI, YANG XIAO DONG, and WANG BING SHAN: 'A fast division technique for constant divisors $2^m(2^n \pm 1)$'. Proceedings of the 1st International Conference on *Computers and Applications*, Beijing, China, June 1985, pp. 715–718

9 LI, R.S.-Y.: 'Fast constant division routines', *IEEE Trans.*, September 1985, **C-34**, (9), pp. 866–889

10 MAGENHEIMER, D., MAY, E., and WHITE, F.: 'Integer multiplication and division on the HP precision architecture'. Proceedings of the 2nd International Conference on *Architecture Support for Programming Languages and Operating Systems*, Oct. 1987, Palo Alto, CA, pp. 90–99

11 JUST, B., MEYER AUF DER HEIDE, F., and WIGDESON, A.: 'On computations with integer division', *Inform. Theor. Appl. (France)*, 1989, **23**, (1), pp. 101–111

12 PETRY, F.E., and SRINIVASAN, P.: 'Division techniques for integers of the form $2^n \pm 1$', *Int. J. Electron.*, 1993, **74**, (5), pp. 659–670

13 SRINIVASAN, P., and PETRY, F.: 'Hardwave design for division by a set of constants for the form $2^n \pm 1$'. Patent disclosure, 1987

14 JACOBSOHN, D.H.: 'A combinatoric division algorithm for fixed integers', *IEEE Trans.*, June 1973, **C-22**, (6), pp. 608–610

15 McCLELLAN, J., and RADER, C.: 'Number theory in digital signal processing' (Prentice-Hall, NJ, 1979)

16 SRINIVASAN, P.: 'Generalized approaches to constant division'. PhD dissertation, Tulane University, Dec. 1988

17 DICKSON, L.E.: 'Modern elementary theory of numbers' (University of Chicago Press, 1939)


## 8 Appendix

### 8.1 Validity of quotient and remainder computation

The summation form of the multiplicative inverse in the division algorithm as in eqn. 9 may be represented as

$$\text{let } \tau_\delta = \sum_{i=1}^{m} h^{i-1} 2^{-in}$$

where $h = \pm 1$, and for $l, n, x \in N$, $(\delta = 2^n - h)$, $m = \lceil l/n \rceil$, and $0 \leqslant x \leqslant 2^l - 1$. For $\delta$, $x$, $q$, $r \in N$ let $x = \delta q + r$, $0 \leqslant r < \delta$. To demonstrate quotient correctness, one must have

$$\lfloor \tau_\delta x \rfloor = q$$

$$\lfloor \tau_\delta x \rfloor = \left\lfloor \frac{x}{\delta}\left(1 - \left(\frac{h}{2^n}\right)^m\right)\right\rfloor = \left\lfloor q + \frac{r}{\delta} + \frac{x}{\delta}\left(\frac{h}{2^n}\right)^m \right\rfloor$$

Since $0 \leqslant x \leqslant 2^l - 1$ and $2^{nm} - 2^{n\lceil l/n \rceil} = 2^{l+\Delta}$ then

$$\frac{x}{\delta 2^{nm}} = \frac{x}{\delta 2^{l+\Delta}} \leqslant \frac{2^l - 1}{\delta 2^{l+\Delta}} < \frac{1}{\delta}$$

Therefore for $h^m = +1$, i.e. $h = +1$ or $m$ even, if $r \geqslant 1$ or $x = 0$ then

$$0 \leqslant \frac{r}{\delta} - \frac{x}{2^{nm}} < 1$$

and

$$\lfloor \tau_\delta x \rfloor = \theta(x) = q$$

if $r = 0$ and $x > 0$ then

$$0 \geqslant \frac{r}{\delta} - \frac{x}{2^{nm}}$$

and

$$\lfloor \tau_\delta x \rfloor = q - 1$$

for $h^m = -1$, i.e. $h = -1$ and $m$ odd,

$$0 \leqslant \frac{r}{\delta} + \frac{x}{2^{nm}} < 1$$

and

$$\lfloor \tau_\delta x \rfloor = \theta(x) = q$$

### 8.2 Remainder computation for divisors of form $2^n \bullet 1$

The dividend $D$ is represented in base $2^n$ digits as $D = 2^0 x_0 + 2^n x_1 + 2^{2n} x_2 + \cdots + 2^{(m-1)n} x_{m-1}$, where $m = \lceil l/n \rceil$. Then, to find $D$ modulo the divisor $\delta$, denoted $|D|\delta$, where $\delta = 2^n \pm 1$, one has

$$D \pmod{2^n - h} = \sum_{i=0}^{m-1} h^i x_i \pmod{2^n - h}$$

The remainder is computed in References 5 and 8 in the least significant column of the computation presented in Fig. 6 and 7, along with an end-around carry. This computation has been summarised as

$$\sum_{i=0}^{m-1} h^i x_i \pmod{2^n} + c_i h$$

where $h = \pm 1$, the divisor $\delta = 2^n - h$, and $c = \{-1, 0, 1\}$. As stated, along with the check for the value $2^n - 1$ in the result, this algorithm computes the remainder.