Radix-2 Division Algorithms with an Over-Redundant Digit Set

Jo Ebergen, *Member, IEEE* and Navaneeth Jamadagni, *Student Member, IEEE*

Abstract—This paper presents a derivation of four radix-2 division algorithms by digit recurrence. Each division algorithm selects a quotient digit from the over-redundant digit set {−2, −1, 0, 1, 2}, and the selection of each quotient digit depends only on the two most-significant digits of the partial remainder in a redundant representation. Two algorithms use a two's complement representation for the partial remainder and carry-save additions, and the other two algorithms use a binary signed-digit representation for the partial remainder and carry-free additions. Three algorithms are novel. The fourth algorithm has been presented before. Results from the synthesized netlists show that two of our fastest algorithms achieve an improvement of 10 percent in latency per iteration over a standard radix-2 SRT algorithm at the cost of 36 percent more power and 50 percent more area.

Index Terms—Signed-digit arithmetic, two's complement arithmetic, carry-save division, carry-free division

1 Introduction

Division is one of the most complex and the slowest arithmetic operations performed in microprocessors. Although division occurs less frequently than other arithmetic operations, having an efficient divider is necessary for a good system performance [1]. There are several division algorithms available to implement in hardware. The digit-recurrence SRT division algorithm is the most frequently implemented algorithm in general purpose processors. A standard radix-2 SRT algorithm retires a quotient digit from the set {−1, 0, 1}. Typically, the selection of a quotient digit relies on the four most significant bits of the partial remainder in a redundant representation. The logic that selects a quotient digit is called the quotient selection logic and it usually appears in the critical path of a divider design. Therefore, simplifying the quotient selection logic potentially leads to a faster divider design which is the main motivation for this work.

1.1 Division Preliminaries

A division algorithm must compute an approximation to $Q = R/D$, where Q is the quotient, D is the divisor and R is the dividend and the initial partial-remainder. We assume that

$$R, D \in [1, 2). \tag{1}$$

For binary representations of R and D, performing the appropriate shift operations before the start of a division algorithm can satisfy these assumptions.

- *J. Ebergen is with Oracle Laboratories, Redwood Shores, CA 94404. E-mail: jo.ebergen@oracle.com.*
- *N. Jamadagni is with Oracle Laboratories, Redwood Shores, CA 94404 and Portland State University, Portland, OR 97201. E-mail: navaja@cecs.pdx.edu.*

Manuscript received 17 Nov. 2013; revised 24 Sept. 2014; accepted 1 Oct. 2014. Date of publication 13 Nov. 2014; date of current version 12 Aug. 2015. Recommended for acceptance by P. R. Schaumont.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2014.2366738

In general, digit-recurrence division algorithms can be described by a recurrence relation

$$r_{n+1} = 2 * r_n - q_n * D, \tag{2}$$

where, n represents the iteration index and r_n is the remainder after the nth iteration with initially $r_0 = R/2$, and q_n is the nth quotient digit selected from the set $\{-1, 0, 1\}$. In each iteration, the algorithm doubles the remainder, then selects a quotient digit q_n, and subtracts $q_n * D$ from r_n. Alternatively, if we start with a different initialization $r_0 = R$, then we can use the recurrence relation

$$r_{n+1} = 2 * (r_n - q_n * D). \tag{3}$$

Here, each repetition step starts with selecting a quotient digit q_n, then subtracting $q_n * D$, and finally doubling the result. The rest of this paper assumes the latter recurrence relation (3) for describing all the algorithms.

Additionally, we require that the error interval of the computed quotient be less than one unit of least precision (ulp), where $ulp = 2^{-L}$ for some $L > 0$. In other words, if q is the computed quotient and the error, ϵ, is given by $\epsilon = q - R/D$, then we require that $\epsilon \in (-ulp/2, ulp/2)$. Alternatively, the error interval may include one of the bounds, but not both bounds. For IEEE single-precision format, $L = 23$, and for IEEE double-precision format, $L = 52$.

2 Related Work

The idea of using an over-redundant quotient-digit set, {−2, −1, 0, 1, 2}, to simplify the quotient selection logic was first presented for a radix-4 SRT algorithm in [2]. In [3], the authors develop a radix-2 SRT algorithm using a signed-digit representation for the partial remainder and an over-redundant quotient-digit set. The quotient selection logic in [3] inspects only the two most-significant digits of the partial remainder in a signed-digit representation to determine an appropriate quotient digit. In [4], Burgess presents a radix-2 Svoboda division algorithm that also inspects only the two most-significant digits of

the partial remainder in a signed-digit representation to retire a quotient digit. This algorithm, however, requires pre-scaling of both divisor and dividend for divisors in the range $[1.5, 2)$.

We derive four radix-2 division algorithms that use an over-redundant quotient-digit set. We denote the four algorithms as A1, A2, B1 and B2. The three algorithms, A1, A2 and B2, are novel and the fourth algorithm, B1, is the same as the one presented in [3]. All four algorithms inspect only the two most-significant bits of the partial remainder to select a quotient digit. Two algorithms use a two's complement representation for the partial-remainder and carry-save additions. The other two algorithms use a signed-digit representation for the partial-remainder and carry-free additions. The major difference between our algorithms and the SRT algorithms in [3], [5], [6] and [7] is the range invariant for the partial remainder. We also use an alternative method to analyze the algorithms presented in this paper and in [3], [5], [6] and [7] by using invariants and highlighting the differences between the algorithms. The analysis method used in this paper is similar to the one described in [8]. This paper extends the algorithm of [8] in several ways: we present three new algorithms and we compare synchronous implementations for all the algorithms discussed in this paper in terms of latency, power, and area. In [9], the authors focus on an asynchronous implementation of the algorithm in [8].

2.1 An Alternative Analysis Method

A conventional method for analyzing a digit-recurrence division algorithm uses a Robertson diagram or a P-D diagram [5], [6]. Both these diagrams show a non-redundant value for the partial remainder, r, even when r is in a redundant form. In an SRT algorithm, r, can be represented in a two's complement carry-save form such that $r = r_s + r_c$ or in a signed-digit carry-free form such that $r = r_c - r_s$, where r_s is called the sum or the parity bits and r_c is called the carry or the majority bits. While the properties of division algorithms can be analyzed at the digit-level [10], when we seek a hardware implementation, the actual encoding of the bits determine the complexity of the circuit. Therefore, we believe that a diagram that clearly shows the redundant representation of the partial remainder will help us design an efficient quotient selection logic. In this paper we present one such diagram and use this diagram to derive and analyze our division algorithms. For clarity, we also show the P-D diagrams for our algorithms.

2.2 An Alternative Range Invariant for the Partial Remainder

The range invariant for the partial remainder depends on the choice of a recurrence relationship and a division algorithm. The SRT algorithms that use the recurrence relation in (2) and a two's complement representation for the partial remainder, have a range invariant of

$$r = r_s + r_c \in [-D, D). \tag{4}$$

The SRT algorithms that use the recurrence relation in (3) and a two's complement representation for the partial remainder, have a range invariant of

$$r = r_s + r_c \in [-2D, 2D). \tag{5}$$

When we use a signed-digit representation for the partial remainder, the range invariant for the partial remainder excludes the lower bound, that is, $r \in (-D, D)$ for recurrence relation in (2) and $r \in (-2D, 2D)$ for recurrence relation in (3). Because we use the recurrence relation in (3) for the algorithms presented in this paper, we consider invariant (5) for SRT algorithms.

Our algorithms, A1, A2, B1 and B2, have a different range invariant. The algorithms that use a two's complement carry-save representation for the partial remainder, $r = r_s + r_c$, have a range invariant of

$$r_s, r_c \in [-2, 2) \quad \text{and} \quad r = r_s + r_c \in [-4, 4). \tag{6}$$

The algorithms that use a signed-digit carry-free representation for the partial remainder, $r = r_c - r_s$ have a range invariant of

$$r_s, r_c \in [0, 4) \quad \text{and} \quad r = r_c - r_s \in (-4, 4). \tag{7}$$

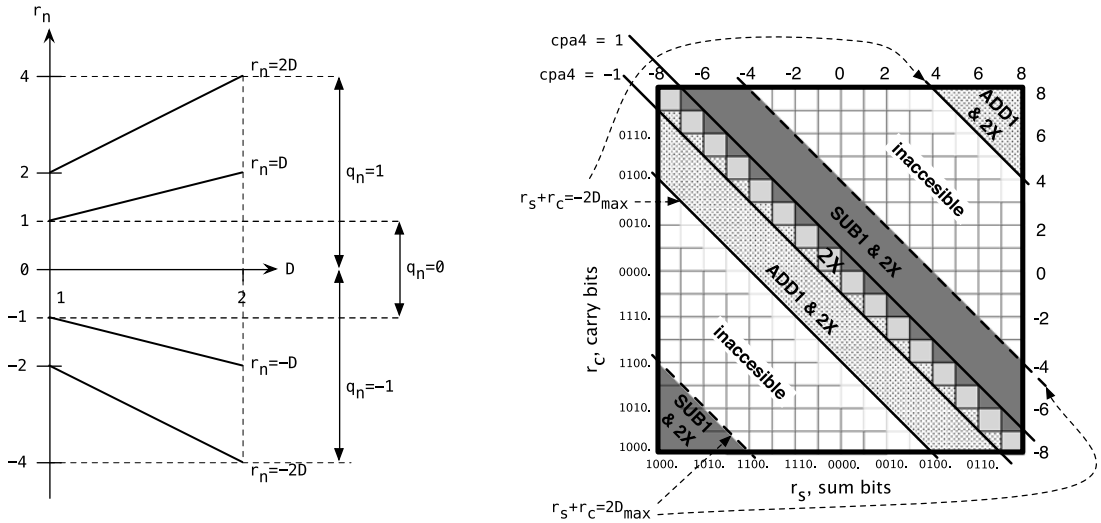
In Section 4, we prove the correctness of our algorithms that use range invariants (6) and (7). In Section 7, we show that algorithms, A1 and B1, also maintain the range invariant (5) for the partial remainder in addition to maintaining invariant (6) or (7).

3 Radix-2 SRT Algorithm

A standard radix-2 SRT algorithm uses a two's complement representation for the partial remainder, r, and carry-save additions (subtractions). The result of a carry-save addition is two numbers, r_s and r_c, whose sum is the actual value. Therefore, $r = r_s + r_c$, where r_s represents the sum or parity bits and r_c represents the carry or majority bits. The standard SRT algorithm has four non-fractional bits and carry-save addition is done modulo 2^4. More information on SRT algorithms can be found in [5], [6] and [7]. The selection of the quotient digit is based on the values of the four most-significant digits of the remainder in carry-save form r_s, r_c. Let $cpa4(r_s, r_c)$ denote the result of a carry-propagate addition of only the four most significant digits of r_s and r_c. The algorithm selects a quotient digit q_n according to the following conditions

$$\begin{aligned} q_n &= 0 \quad \text{if } cpa4(r_s, r_c) = -1 \\ q_n &= +1 \quad \text{if } cpa4(r_s, r_c) > -1 \\ q_n &= -1 \quad \text{if } cpa4(r_s, r_c) < -1 \end{aligned}$$

Figs. 1a and 1b show the quotient selection function in a conventional P-D diagram and in the r_s, r_c plane respectively. In Fig. 1a, the *x*-axis represents the value of the divisor and the *y*-axis represents the value of the partial remainder. In Fig. 1b, the *x*-axis represents the value of the sum bits and is labeled with both the absolute value of r_s (top) as well as the four non-fractional bits of the two's complement representation (bottom). The *y*-axis represents the value of the carry bits and is labeled with both the absolute value of r_c (right) as well as the four non-fractional bits of the two's complement representation (left). A point in this figure has coordinates (r_s, r_c).



(a) P-D plot for the standard radix-2 SRT algorithm.

(b) Standard radix-2 SRT algorithm in (r_s, r_c) plane

Fig. 1. Standard radix-2 SRT algorithm: Figure (a) shows the P-D plot for a standard radix-2 SRT algorithm. The x-axis represents the value of the divisor and y-axis represents the value of the partial remainder. Figure (b) illustrates the quotient selection areas for a standard radix-2 SRT division in the (r_s, r_c) plane using only the four most-significant bits. In the (r_s, r_c) plane, each point has coordinates (r_s, r_c). Each diagonal line $r_s + r_c = r$ modulo 2^4 represents a set of points with different r_s and r_c values but the same remainder value. Addition is modulo 2^4, so diagonal bands wrap around the square. For radix-2 SRT division, the remainder $r_s + r_c$ remains within the range $[-2D, 2D)$. In the figure $D_{max} = 2 - ulp$.

Each diagonal line $r_s + r_c = r$ modulo 2^4 represents a set of points with the same remainder value. The area labeled 2X is the area where $cpa4(r_s, r_c) = -1$. For every remainder in this area, the SRT algorithm selects the quotient digit 0 and performs a doubling. The area labeled SUB1&2X is the area where $cpa4(r_s, r_c) > -1$. For every remainder in this area, the SRT algorithm selects quotient digit 1 and performs a subtraction with D followed by a doubling. The area labeled ADD1&2X is the area where $cpa4(r_s, r_c) < -1$. For every remainder in this area, the SRT algorithm selects quotient digit -1 and performs an addition with D followed by a doubling. Because addition is calculated modulo 2^4, the diagonal bands wrap around the square.

Because the SRT algorithm satisfies the invariant $r_s + r_c \in [-2D, 2D)$, only the shaded areas are accessible. There are large inaccessible areas. In fact, at least half the area is inaccessible. These large inaccessible areas suggest that there may be more efficient quotient selections that utilize fewer digits. In the following sections we derive such quotient selection algorithms.

4 The Invariants and Termination

We use recurrence relations and invariants to prove the correctness of our division algorithms and calculate the error in the computed quotient. In this section we make explicit which invariants we use.

The formula

$$Q * D = R \tag{8}$$

expresses the desired relation between Q, D, and R, where Q is the exact quotient. In our algorithms, we use lowercase variables q and r, where q represents the quotient calculated 'thus far,' and r represents the remainder calculated 'thus far.' The invariant for all the variables is as follows:

$$q * D + 2^{-n} * r = R. \tag{9}$$

In this invariant, n is the iteration index and $q_n * 2^n$ is added to q in the nth iteration, where q_n is the quotient digit selected in nth iteration.

We look for a number of program statements for the program variables q, r, and c that establish or maintain invariant (9). Once we have these program statements, we can then combine the statements in various ways to obtain a division algorithm.

The initialization `q=0; r=R; n=0` establishes invariant (9). Any quotient digit from an over-redundant digit-set $\{-2, -1, 0, 1, 2\}$ and the recurrence equation (3) will maintain the invariant (9). The challenge is to choose a quotient digit that will maintain the range invariant (6) if the partial remainder is in a two's complement representation or the range invariant (7) if the partial remainder is in a signed-digit representation.

TABLE 1
Labels to Denote the Choice of a Quotient Digit and the Statements Executed

Label	Quotient Digit, q_n	Statements executed
ADD2 & 2X	-2	$r = 2 * (r + 2D);$ $q = q - 2 * 2^{-n};$ $n = n + 1$
ADD1 & 2X	-1	$r = 2 * (r + D);$ $q = q - 1 * 2^{-n};$ $n = n + 1$
2X	0	$r = 2 * r;$ $q = q - 0 * 2^{-n};$ $n = n + 1$
SUB1 & 2X	$+1$	$r = 2 * (r - D);$ $q = q + 1 * 2^{-n};$ $n = n + 1$
SUB2 & 2X	$+2$	$r = 2 * (r - 2D);$ $q = q + 2 * 2^{-n};$ $n = n + 1$

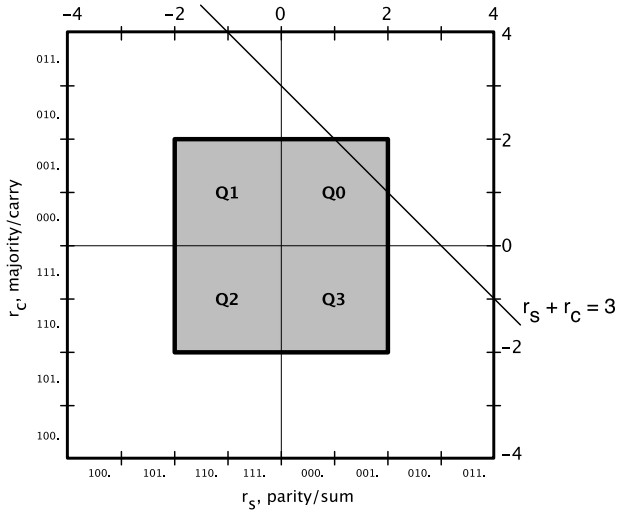


Fig. 2. The area for partial remainders (r_s, r_c) where the value of the remainder r is given by $r = r_s + r_c$. The partial remainders r_s and r_c are in a two's complement representation. The points (r_s, r_c) on a diagonal line, like $r_s + r_c = 3$, can have different values for r_s and r_c but have the same remainder value. The center square, Q0∪Q1∪Q2∪Q3, satisfies range invariant (6).

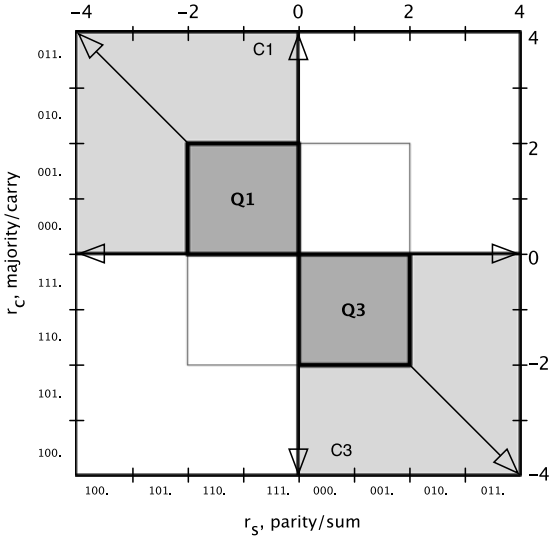


Fig. 3. The effect of doubling all points in Q1 or Q3.

We use labels to denote the choice of a quotient digit and the statements executed to update the partial remainder (according to (3)), quotient, and the iteration index. Table 1 lists the labels corresponding to the choice of a quotient digit and the statements executed.

Now we need to make sure that the error in the computed quotient is small enough. Using invariant (9) we can express the error, ϵ, in the computed quotient as

$$\epsilon = R/D - q = 2^{-n} * r/D. \tag{10}$$

Consequently, by requiring that $2^{-n} * r/D \in [-ulp/2, ulp/2)$, we require that the computed quotient q has an error interval of length less than ulp. For a given ulp, the condition $2^{-n} * r/D \in [-ulp/2, ulp/2)$ translates into a condition which determines the value of n, and a condition for the range of the remainder r, which determines the value of r/D.

For example, if we consider the range invariant (5), the condition $2^{-n} * r/D \in [-ulp/2, ulp/2)$ translates into the condition $2^{-n} * 2 \leq ulp/2$, where $ulp = 2^{-L}$. Thus, the termination condition becomes $n \geq L + 2$. The range invariant may also exclude the lower bound, that is, $(-2D, 2D)$ instead of $[-2D, 2D)$

If we consider the range invariants (6) or (7), the termination condition $2^{-n} * r/D \in [-ulp/2, ulp/2)$ translates into the condition $2^{-n} * (4/D) \leq ulp/2$, where $ulp = 2^{-L}$ and $D \in [1, 2)$. In this case the termination condition becomes $n \geq L + 3$. Consequently a division algorithm using the range invariants (6) or (7) requires one more iteration to obtain the same accuracy than a division algorithm using the range invariant (5).

5 Two's Complement Implementation and Carry-Save Addition

For the derivation of our first set of algorithms we analyze what happens with doublings and additions in the (r_s, r_c) plane, when numbers are represented in two's complement and additions are implemented by means of carry-save additions. For clarity, we also provide the P-D diagrams for our algorithms. A two's complement representation of m non-fractional bits can represent numbers in the range $[-2^{m-1}, 2^{m-1})$. Note that the lower bound is inclusive while the upper bound is exclusive. Adding and subtracting numbers in two's complement arithmetic can be done by means of carry-save addition modulo 2^m.

We observe that in order to represent the initial value of $D \in [1, 2)$ and partial remainders r_s and r_c in the range $[-2, 2)$ we only need two non-fractional bits rather than four in the SRT algorithm of Fig. 1b. Let us look at the effect of the addition and doubling operations on all points that satisfy range invariant (6). This is the bold center-square, (Q0∪Q1∪Q2∪Q3), in Fig. 2. To illustrate the effect of an addition and a doubling operation we use the (r_s, r_c) plane and consider three non-fractional bits. An addition or a doubling operation can yield a point inside the center square or outside the center square. For the points that are outside the center square we introduce an extra operation that returns the point to the center square while maintaining invariant (9).

5.1 Doublings and Translations

Fig. 3 illustrates the effect of doubling any point in the small square Q1. A doubling expands the smaller square Q1 into the larger square C1. Notice that in square C1, $r \in [-4, 4)$ but $r_s \in [-4, 0)$ and $r_c \in [0, 4)$ which violates invariant (6). To maintain invariant (6), we need to bring back the points in the square C1 to the center square. To bring back the points in the square C1 to the center square, we perform a translation over $(2, -2)$, as illustrated in Fig. 4. Translation over $(2, -2)$ is implemented by adding 2 to r_s and subtracting 2 from r_c. Note that the translation keeps the value of $r_s + r_c$ unchanged. In fact, any translation of a point (r_s, r_c) over distance $(t, -t)$ for any number t maintains the value of $r_s + r_c$. Note that because translations involve addition and subtraction with a constant, the translations can be implemented by a simple recoding of r_s and r_c.

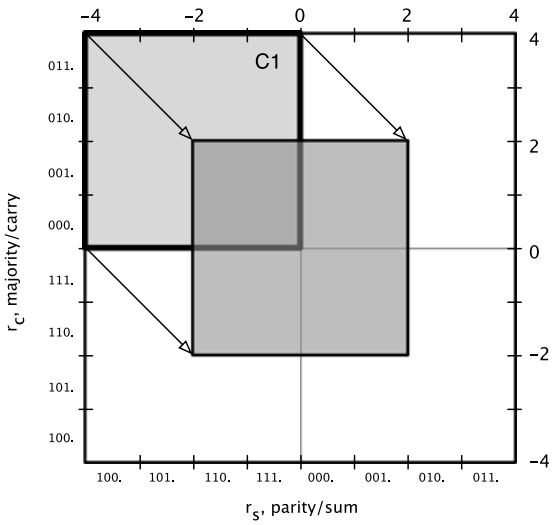


Fig. 4. The effect of translating C1 over (2, −2).

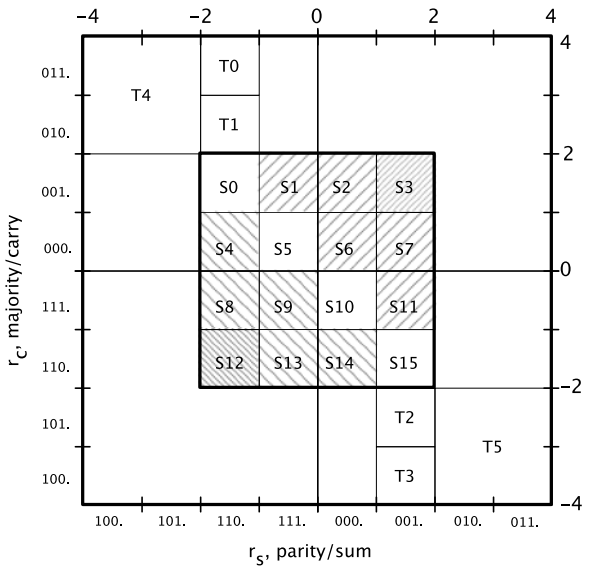


Fig. 5. The effects of carry-save additions and subtractions with D. The division algorithms can perform carry-save additions or subtractions in the shaded squares.

Any doubling of square Q1 followed by a translation over $(2, -2)$ in effect expands the square Q1 to the center square. Similarly, doubling of square Q3 followed by a translation over $(-2, 2)$ expands the square Q3 to the center square. In both cases, the doubling followed by the translation maintain invariant (9) and range invariant (6).

How do we implement these doublings and translations? Doublings can be implemented by left shifting the partial remainders r_s and r_c by one position. The translations over $(2, -2)$ and $(-2, 2)$ can be implemented by a simple recoding of the most-significant bits of r_s and r_c as follows.

$$\begin{array}{lcl} 10 & \rightarrow & 11 \\ 11 & \rightarrow & 00 \\ 01 & \rightarrow & 00 \\ 00 & \rightarrow & 11 \end{array}$$

Notice that the second-most significant bit in each case changes and the most significant bit is a copy of the second-most significant bit.

If all operations start and end in the center square, we can apply some simplifications to the doubling and translation implementations. First, because the two most-significant bits of r_s and r_c are always the same in the center square, we may omit the most significant bit. Second, if we omit the most significant bit, a doubling followed by a translation of a point (r_s, r_c) in the center square simply becomes a left shift by one followed by an inversion of the most significant bit of both r_s and r_c. Because of the extra inversion of the most significant bit, we refer to a doubling followed by a translation as a 2X* operation.

Here is an example of a doubling followed by a translation operation (2X*). Consider a point (r_s, r_c) with three non-fractional bits $(000.u, 110.v)$, where u and v are some bit-sequences. Doubling the point $(000.u, 110.v)$ yields a point $(00?.u', 10?.v')$, where u' and v' are u and v left shifted by 1 position respectively, and $?$ represents a bit value of either 1 or 0 corresponding to the most-significant bit of u or v. Translation of the point $(00?.u', 10?.v')$ yields a point $(11?.u', 11?.v')$ in square Q2 of Fig. 2.

5.2 Carry-Save Addition

Let us look at what happens to a point in the center square when a subtraction with D occurs. We assume that r_s represents the sum bits and r_c represents the carry bits after a carry-save addition. Fig. 5 shows the (r_s, r_c) plane, where we have partitioned the center square in small squares. What happens when we subtract D from a point (r_s, r_c) in each of the small squares? Here is the calculation for a point in square S1 where we consider only the three most significant bits of each number. In two's complement representation $D = 001.x$, for some bit vector x, thus $-D$ is represented by the bit-wise complement of D plus ulp, i.e., $-D = 110.y + ulp$, where y is the bit-wise complement of x. For square S1 we get the following calculation. Note that we shift the majority bits one position to the left to correspond with 'carrying' this bit to the next more significant bit position,

$$\begin{array}{rl} \mathtt{r_s} & \mathtt{111} \\ \mathtt{r_c} & \mathtt{001} \\ -\mathtt{D} & \mathtt{110.y + ulp} \\ \hline \mathtt{sum} & \mathtt{000} \\ \mathtt{carry} & \mathtt{11?} \end{array}$$

For square S2 we get the following calculation:

$$\begin{array}{rl} \mathtt{r_s} & \mathtt{000} \\ \mathtt{r_c} & \mathtt{001} \\ -\mathtt{D} & \mathtt{110.y + ulp} \\ \hline \mathtt{sum} & \mathtt{111} \\ \mathtt{carry} & \mathtt{00?} \end{array}$$

Consequently, subtracting D from a point in square S1 yields a point in squares S10∪S14. Subtracting D from a point in the square S2 yields a point in squares S1∪S5. Because carry-save addition is symmetrical in r_s and r_c, subtracting D from a point in square S11 also yields a point in squares S10∪S14.

TABLE 2
The Effect of Subtracting or Adding D

square	$-D$	square	$+D$
S0	T2∪T3	S0	T0∪T1
S1	S10∪S14	S4	S1∪S5
S2	S1∪S5	S5	T0∪T1
S3	T0∪T1	S8	S10∪S14
S5	T2∪T3	S9	S11∪S15
S6	S0∪S4	S10	T0∪T1
S7	S1∪S5	S12	T2∪T3
S10	T2∪T3	S13	S10∪S14
S11	S10∪S14	S14	S1∪S5
S15	T2∪T3	S15	T0∪T1

Similarly, addition of D to a point in square S4 yields a point in squares S1∪S5, and addition of D to any point in square S8 or S13 yields a point in squares S10∪S14.

Table 2 gives a summary of adding and subtracting D from most small squares. Observe that a subtraction of D from points in the squares S1, S2, S6, S7, and S11 always ends in the squares Q1 or Q3 of Fig. 2. This means that any such point can subsequently undergo a doubling and a translation (i.e., a 2X* operation) and land in the center square.

Square S3 is different. Subtraction of D from points in square S3 yields a point in squares T0∪T1. Translating a point in T0∪T1 yields a point in S2∪S6, where a point must undergo another subtraction before a doubling. Instead, let us calculate what happens when we subtract $2D$, instead of D, from any point in S3. First, recall that in a two's complement representation with 3 non-fractional bits $D = 001.bx$ for some bit b and bit vector x. Thus $2D = 01b.x0$, and $-2D$ is represented by the bit-wise complement of $2D$ plus ulp, that is, $-2D = 10d.y + ulp$, where d is the bit complement of b and y is the bit-wise complement of $x0$,

$$\begin{array}{rl} \mathtt{r_s} & 001 \\ \mathtt{r_c} & 001 \\ -2\mathtt{D} & 110.\mathtt{y} + \mathtt{ulp} \\ \hline \mathtt{sum} & 10? \\ \mathtt{carry} & 01? \end{array}$$

As a consequence, subtracting $2D$ from any point in square S3 yields a point in square T4 of Fig. 5.

We can translate each point in T4 over $(2, -2)$ and the final result lands in square Q1 of Fig. 2. Subsequently, for each point in Q1 we can perform a doubling and translation (ie a 2X* operation) and obtain a point in the center square again.

Similar remarks can be made for the additions of D or $2D$ to points in squares S4, S8, S9, S12, S13 and S14 of Fig. 5. Addition of D to points in squares S4, S8, S9, S13, and S14 always yields a point inside squares Q1 or Q3 of Fig. 2. This means that any such point can undergo a doubling and a translation and again land in the center square.

Addition of D to any point in square S12 yields a point in square T2∪T3. Translating a point in square T2∪T3 yields a point in S9∪S13 where a point must undergo another addition before a doubling. Adding $2D$, however, to any point in square S12 yields a point in square T5. Furthermore, T5 can be translated over $(-2, 2)$ to obtain square Q3 in the center square of Fig. 2. Each point in Q3 can be doubled and translated to obtain a point in the center square again.

In summary, for each small square in the center square we have found a sequence of operations that maintain invariants (9) and (6). Each sequence of operations ends with a doubling and a translation, which may be preceded by a subtraction or addition of D or $2D$. For example, for a point in small square S12, the operations are an addition of 2D, followed by a translation, then a doubling and a translation. Some squares even have two possible sequences of operations that maintain invariant (9) and (6) (keeps a point in the center square of Fig. 2). For example, for points in square S4 the sequence of operations may be a doubling followed by a translation (2X*) or an addition of D followed by a doubling and a translation. Squares S1, S11, and S14 also have two possible sequences of operations.

If we confine ourselves to the points in the center square, each of these operations is easy to implement. Because we deal with the points only in the center square, we can omit the most-significant bit and consider the remaining two non-fractional bits. The operations are implemented as follows.

- Each addition is simply a carry-save addition in two's complement arithmetic.
- Each doubling is a left shift of both r_s and r_c by one position.
- Each translation is an inversion of the most-significant bit for r_s and r_c.

Note that a translation followed by a doubling and then another translation is the same as a doubling followed by a translation, because each doubling throws away the most significant bit.

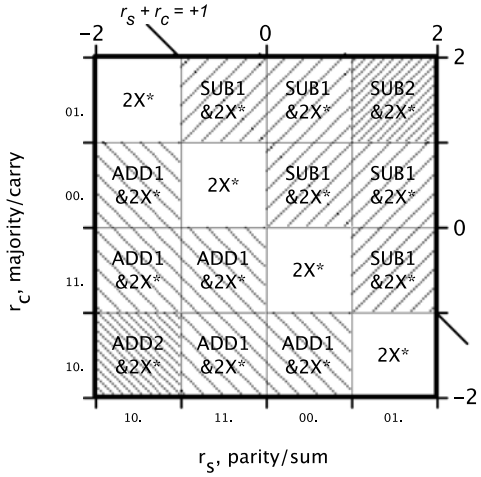
6 Putting It All Together

With the analysis of the previous section, we can put together various division algorithms. For each division algorithm we specify what sequence of operations must be performed on the points (r_s, r_c) in each of the small squares, S0 to S15, in Fig. 5. There are five sequences of operations to choose from which are described as follows:

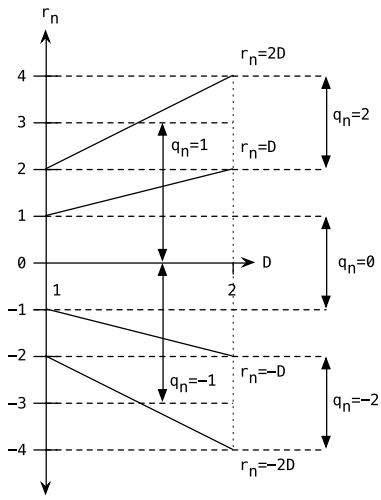
- 2X*: A doubling followed by a translation. The selected quotient digit is 0.
- SUB1&2X*: A subtraction of D followed by a doubling and then a translation. The selected quotient digit is +1.
- SUB2&2X*: A subtraction of $2D$ followed by a doubling and then a translation. The selected quotient digit is +2.
- ADD1&2X*: An addition of D followed by a doubling and then a translation. The selected quotient digit is -1.
- ADD2&2X*: An addition of $2D$ followed by a doubling and then a translation. The selected quotient digit is -2.

Recall that a translation (over (2, -2) or (-2, 2)) is implemented by an inversion of the most-significant bit.

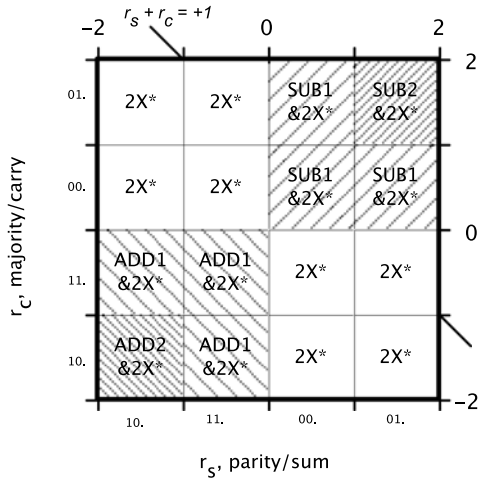
Fig. 6 illustrates two possible choices for a division algorithm. Other algorithms can be derived by making different choices for the squares S1, S4, S11, and S14. Algorithms A1



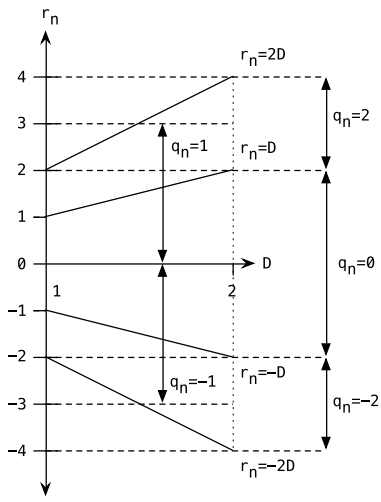
(a) Algorithm A1 in r_s, r_c plane



(b) P-D Plot for Algorithm A1



(c) Algorithm A2 in r_s, r_c plane



(d) P-D Plot for Algorithm A2

Fig. 6. Algorithms A1 and A2: Figures (a) and (c) show the quotient selection function for algorithms A1 and A2 in the (r_s, r_c) plane respectively. Figures (a) and (b) show P-D plots for algorithms A1 and A2 respectively.

and A2, however, are the most symmetric choices. The selection of a quotient digit relies on only the two most-significant bits of r_s and r_c. Algorithm A2 has a simpler selection logic than Algorithm A1, which can lead to a faster divider implementation.

Thus far, we have shown that Algorithms A1 and A2 maintain the range invariant (6) for the partial remainder. In the next section, we show that Algorithm A1 also maintains the range invariant (5) for the partial remainder. As discussed in Section 4, because algorithm A1 maintains the range invariant (6), algorithm A1 must execute at least $L + 3$ iterations. Because algorithm A2 maintains the range invariant (5), algorithm A2 must execute at least $L + 2$ iterations.

7 A Different Range Invariant

In this section, we prove that Algorithm A1 maintains the range invariant (5):

$$r_s + r_c \in [-2D, 2D).$$

This means that Algorithm A1 needs one fewer iteration than Algorithm A2 to satisfy the accuracy requirement for the computed quotient.

First, we observe that the range invariant (5) holds after initialization $r_s = R; r_c = 0$ for $R, D \in [1, 2)$.

Second, we observe that each of the operations ADD2&2X*, ADD1&2X*, SUB1&2X*, and SUB2&2X* maintains range invariant (5). Here is a proof for the sequence of operations SUB1&2X* and SUB2&2X*. Assume that the invariant (5) holds before each of those sequences of operations. Hence, in the regions of Fig. 6a where Algorithm A1 executes the sequence of operations SUB1&2X* we have

$$r_s + r_c \in [0, 2D).$$

After subtracting D from $r_s + r_c$ and doubling r_s and r_c, we have

$$r_s + r_c \in [-2D, 2D),$$

which is our range invariant (5).

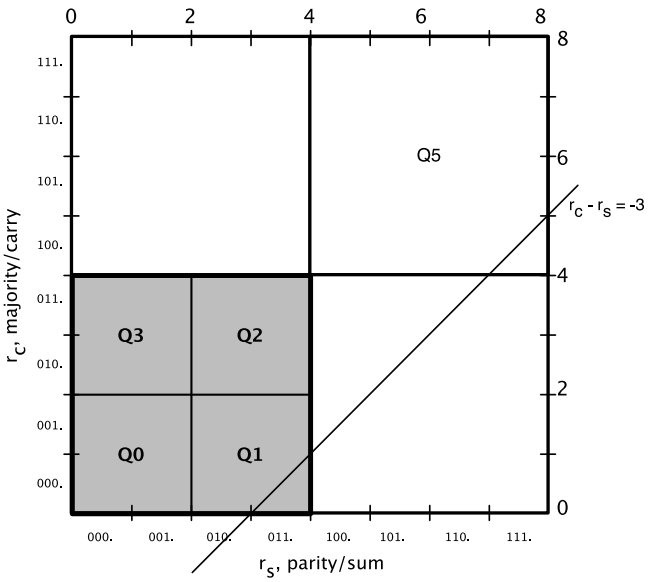


Fig. 7. The area for the partial remainders (r_s, r_c), where the value of the partial remainder r is given by $r = r_c - r_s$. All points (r_s, r_c) on a diagonal line, like $r_c - r_s = -3$, represent the same remainder value. The bottom-left square consisting of Q0, Q1, Q2, and Q3 squares satisfies the range invariant (7).

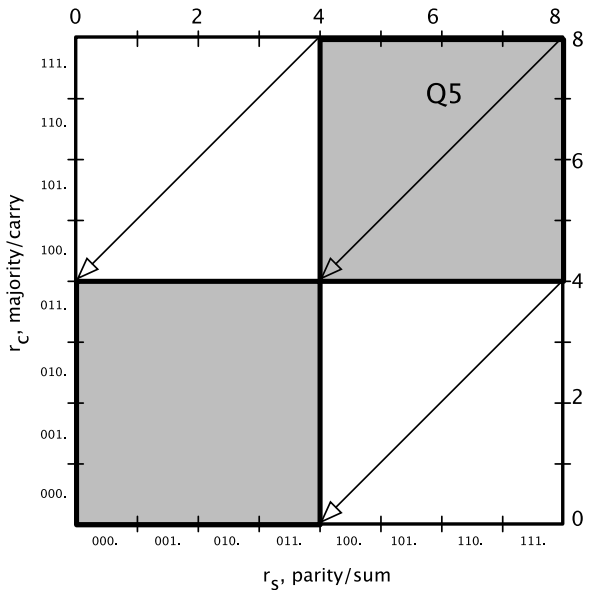


Fig. 8. Translation of square Q5 over $(-4, -4)$.

In the regions of Fig. 6a where Algorithm A1 executes the operations SUB2&2X*, we have before the operations

$$r_s + r_c \in [2, 2D).$$

After subtracting $2D$ from $r_s + r_c$ and doubling r_s and r_c, we have

$$r_s + r_c \in [2(2 - 2D), 0).$$

Furthermore, we have $2(2 - 2D) = (4 - 2D) - 2D > -2D$, because $4 - 2D > 0$ for $D \in [1, 2)$. Consequently, after the operations SUB2&2X*, we have $r_s + r_c \in [-2D, 2D)$.

In a similar way, we can prove that the operations ADD1&2X* and ADD2&2X* maintain invariant (5). Finally we prove that also the operations 2X* maintain invariant (5). Observe that if Algorithm A1 executes the operation 2X* then r_s, r_c are in one of the diagonal squares, which means that $r_s + r_c \in [-1, 1)$. Because $D \in [1, 2)$, we have $r_s + r_c \in [-D, D)$. Consequently, after the doubling operation we have $r_s + r_c \in [-2D, 2D)$, our range invariant (5).

8 BSD Representations and Carry-Free Additions

Instead of using a two's complement representation for the partial remainder, we can use a binary signed digit (BSD) representation. In a BSD representation, the partial remainder r is represented by a vector of signed bits from the set $\{-1, 0, 1\}$. Instead of using a single vector of signed bits, we use two vectors of unsigned bits r_s and r_c, such that $r = r_c - r_s$. Please see [6] for more details on BSD representation and carry-free addition.

For the BSD representation of the remainder, we use the range invariant (7) The partial remainder $r = r_c - r_s$ is in the range $(-4, 4)$. This range is similar to the range $[-4, 4)$ for the remainder r when using a two's complement representation. For the BSD representation, however, both bounds are excluded.

The bottom-left bold square, Q0∪Q1∪Q2∪Q3, in Fig. 7 consists of all numbers (r_s, r_c) satisfying range invariant (7).

We call vector r_c the carry and vector r_s the sum, where the sum vector has a negative weight.

To see what happens when we perform doublings and additions on points satisfying range invariant (7), we first look at a larger square in Fig. 7, where each binary vector is represented with one extra digit at the most-significant position. We are interested in a series of operations that takes a point in the bottom-left bold square and returns a point in the bottom-left bold square. As with the two's complement representations, the operations must end with a doubling and may be preceded by an addition or subtraction. Each sequence of operations must maintain invariant (9).

8.1 Doublings and Translations

The effects of a doubling is straightforward. Doubling a point in square Q0 of Fig. 7 returns a point in the bottom-left square, Q0∪Q1∪Q2∪Q3. Doubling of a point in square Q2 returns a point in the top-right square Q5. Each point in Q5 can be translated back to the bottom-left square by a translation over $(-4, -4)$, as illustrated in Fig. 8. Any translation over $(-t, -t)$ leaves the value of $r = r_c - r_s$ invariant and thus each translation maintains invariant (9)). Furthermore, a translation over $(-4, -4)$ for any point in Q5 is easy to implement by inverting the most-significant bit (bit position with weight 2^3).

8.2 Additions

Squares Q1 and Q3 in Fig. 7 require more operations than just doublings and translations. We analyze the effect of additions to the points in each of the small squares, S0 to S15, in Fig. 9.

We implement the addition $r + z$ of a remainder $r = r_c - r_s$ and a choice $z \in \{-2D, -D, D, 2D\}$ by means of a carry-free addition [6]. As shown in [6], we have

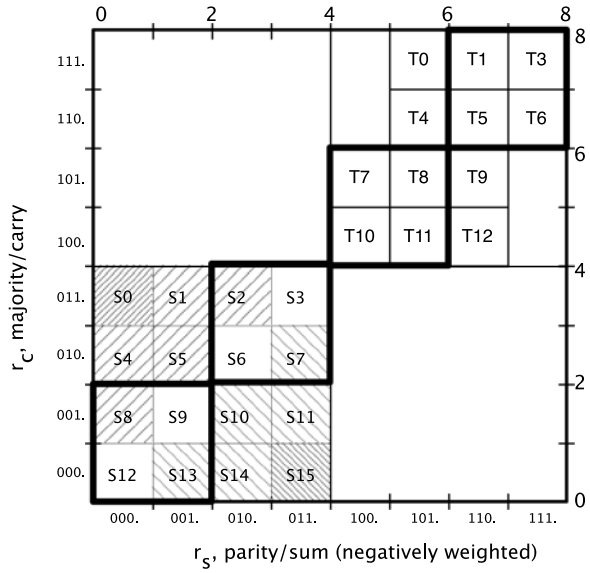


Fig. 9. The effect of carry-free additions and subtractions with D or $2D$. The division algorithms can perform carry-free additions or subtraction in the shaded squares.

$$r_c - r_s + z = BSDcarry - BSDsum,$$

where $BSDcarry$ and $BSDsum$ are the result of a carry-save addition of $\neg r_s, r_c$, and z, with an inversion of the parity result:

$$BSDcarry = 2 * majority(\neg r_s, r_c, z),$$
$$BSDsum = \neg parity(\neg r_s, r_c, z).$$

The majority and parity functions, forming the carry-free addition, can be done modulo 2^m if there are m non-fractional bits. In our case, $m = 3$. This implementation of carry-free addition applies only if we take a two's complement representation for z [6]. Thus $-D$ and $-2D$ can be represented by $-D = 110.y + ulp$ and $-2D = 10b.y + ulp$ respectively, for some bit vector y and bit b.

For example, for subtracting D from a point in square S1 we get the following carry-free addition:

$$\begin{array}{rr} \mathtt{r_s} & 001 \\ \mathtt{r_c} & 011 \\ -\mathtt{D} & \mathtt{110.y + ulp} \\ \hline \mathtt{sum} & 100 \\ \mathtt{carry} & \mathtt{10?} \end{array}$$

These values for *BSDsum* and *BSDcarry* correspond to a point (r_s, r_c) in squares T7 or T10. For adding D to a point in square S13 we get the following carry-free addition:

$$\begin{array}{rr} \mathtt{r_s} & 001 \\ \mathtt{r_c} & 000 \\ +\mathtt{D} & \mathtt{001.y} \\ \hline \mathtt{BSDsum} & 000 \\ \mathtt{BSDcarry} & \mathtt{00?} \end{array}$$

These values for *BSDsum* and *BSDcarry* correspond to a point in squares S8 or S12.

TABLE 3
The Effect of Adding or Subtracting D

square	$-D$	square	$+D$
S0	T0∪T4	S3	S10∪S14
S1	T7∪T10	S6	S10∪S14
S2	T3∪T6	S7	S8∪S12
S3	T9∪T12	S9	S10∪S14
S4	T7∪T10	S10	S2∪S6
S5	T8∪T10	S11	S3∪S7
S6	T9∪T12	S12	S1∪S5
S8	T3∪T6	S13	S8∪S12
S9	T9∪T12	S14	S3∪S7
S12	T9∪T12	S15	S10∪S14

Table 3 shows the results of carry-free addition in squares S0 to S15 in Fig. 9. All subtractions of D from squares S1, S2, S4, S5, and S8 yield points in small squares T3, T6, T7, T8, T10, or T11 which can be translated, doubled, and translated again to return a point in one of the S0 to S15 squares.

Subtracting D from any point in square S0 yields a point in square T0 or T4. Translating a point in the squares T0 or T4 yields a point in squares S1 or S5 where the point must undergo another subtraction. Therefore, instead of subtracting D, we subtract $2D$ from a point in S0 using carry-free addition

$$\begin{array}{rr} \mathtt{rs} & \mathtt{000} \\ \mathtt{rc} & \mathtt{011} \\ -\mathtt{2D} & \mathtt{10?.y + ulp} \\ \hline \mathtt{BSDsum} & \mathtt{11?} \\ \mathtt{BSDcarry} & \mathtt{11?} \end{array}$$

The resulting values for *BSDsum* and *BSDcarry* correspond a point in squares T1, T3, T5, or T6. Points in these squares can be translated, doubled and translated again to return a point in one of the S squares.

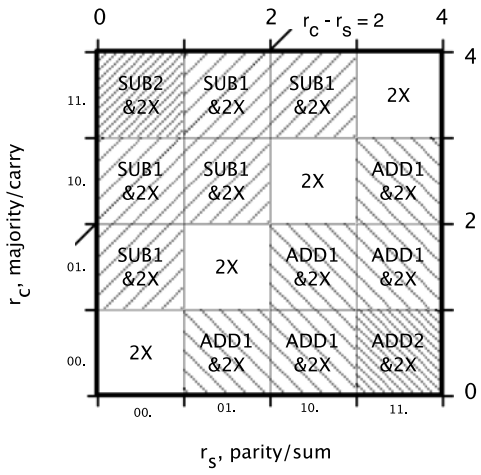
All additions of D to points in squares S7, S10, S11, S13, and S14 yield points in small squares S2, S3, S6, S7, S8, or S12. Points in these squares can all be doubled and translated to return a point in one of the S0 to S15 squares.

Adding D to a point in square S15 yields a point in squares S10 or S14 where the point must undergo another addition rather than a doubling. Adding $2D$, however, to any point in square S15 returns a point in squares S8, S9, S12, or S13, which can be doubled and translated to remain in one of the S0 to S15 squares.

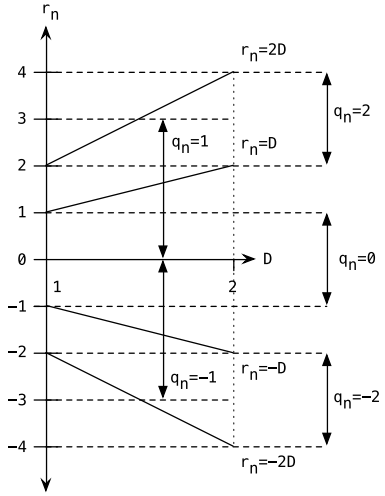
With the analysis of the doublings, translations, and additions, we can put together a number of division algorithms based on the BSD representation for the partial remainder and carry-free additions.

Before giving the five possible choices for sequences of operations, we consider a few simplifications. Because each set of operations in our division algorithm takes a point in one of the S0 to S15 squares and returns a point in one of the S0 to S15 squares, we omit the most-significant bit, which is always 0, and use only two non-fractional bits. The omission of the most-significant bit simplifies the implementation of the translation to an operation that is implemented automatically.

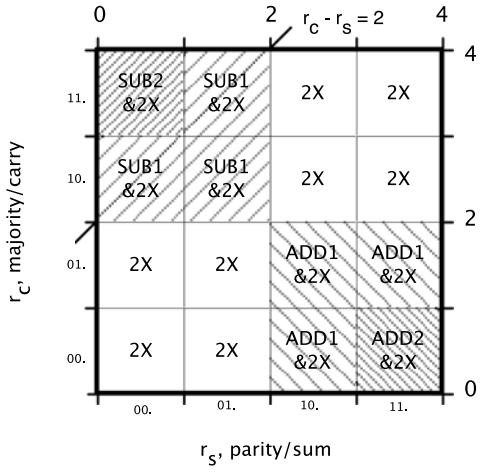
The five choices for the sequences of operations are: 2X, SUB1&2X, SUB2&2X, ADD1&2X and ADD2&2X. The



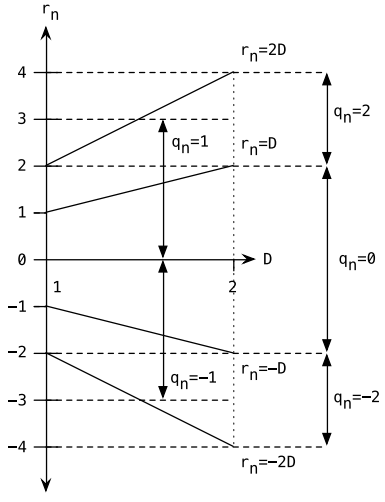
(a) Algorithm B1 in (r_s, r_c) plane.



(b) P-D Plot for Algorithm B1.



(c) Algorithm B2 in (r_s, r_c) plane.



(d) P-D Plot for Algorithm B2.

Fig. 10. Two division algorithms, B1 and B2, based on BSD representations and carry-free additions. Figures (a) and (c) illustrate the quotient selection function of algorithms B1 and B2 in (r_s, r_c) plane. Figures (b) and (d) show the quotient selection function of algorithm B1 and B2 using P-D plots respectively. Algorithm B1 is the same algorithm as presented in [3].

quotient digit selected and the statements executed for each of these operation is listed in Table 1. Furthermore, each of these operations maintain invariant (9) and range invariant (7).

We can compose a division algorithm by choosing one sequence of operations for each small square S0 through S15. For the squares S2, S7, S8, and S13 there are two choices for selecting a quotient digit. For the squares S2 and S8 the two choices are as follows: selecting a quotient digit 0 and performing a 2X operation or selecting a 1 and performing a SUB1&2X operation on the partial remainder. For the squares S7 and S13 the two choices are as follows: selecting a quotient digit 0 and performing a 2X operation or selecting a -1 and performing a ADD1&2X operation on the partial remainder. For all other squares there is only one choice for selecting a quotient digit. The two most symmetric algorithms appear in Fig. 10. We characterize each algorithm by the choice of sequence of operations that is performed for each point in a small square. The operations are similar to the operations for a two's complement representation and carry-save additions.

Algorithms B1 and B2 in Fig. 10 satisfy the range invariant (7). Algorithm B1, also satisfies the range invariant $r = r_c - r_s \in (-2D, 2D)$. The proof that Algorithm B1 satisfies the range invariant $r \in (-2D, 2D)$ is essentially the same as in Section 7. Therefore, algorithms B1 and B2 require at least $L + 2$ and $L + 3$ iterations, respectively, to terminate with an error $\epsilon \in (-ulp/2, ulp/2)$.

Algorithm B1 in Fig. 10a is the same algorithm as presented in [3]. Note that in [3], the authors use the recurrence relation in (2) and hence the authors use the range invariant $r \in (-D, D)$ for the partial remainder. Because we have considered the recurrence relation in (3) throughout this paper, the range invariant for the partial remainder in [3] translates to $r \in (-2D, 2D)$.

9 Implementation Results

In this section we compare the radix-2 division algorithms presented in this paper for latency per iteration, power and area. While the implementation of an algorithm can be

TABLE 4
Comparison of the Five Radix-2 Algorithms Discussed in This Paper

Algorithm	Partial remainder representation	Partial remainder range	Type of Adder	Quotient digit set	Latency per iteration in ps	Avg. power per iteration in mW	Area in μm^2
SRT	Two's Complement	$r \in [-2D, 2D)$	Carry-save	{−1,0,1}	250	7.60	7,900
A1	Two's Complement	$r \in [-2D, 2D)$	Carry-save	{−2,−1,0,1,2}	240	10.08	11,376
A2	Two's Complement	$r \in [-4, 4)$	Carry-save	{−2,−1,0,1,2}	225	10.35	11,676
B1	Signed-Digit	$r \in (-2D, 2D)$	Carry-free	{−2,−1,0,1,2}	240	10.08	11376
B2	Signed-Digit	$r \in (-4, 4)$	Carry-free	{−2,−1,0,1,2}	225	10.35	11,676

further optimized for low-latency or low-power [11], [12] and [13], we focus on comparing the algorithms in the same design environment. For comparison, we synthesized the behavioral verilog code for all the algorithms using Synopsys Design Compiler and a production quality TSMC 40 nm standard cell library. To estimate the latency per iteration and area, we used Synopsys Design Compiler's static timing analysis engine and for power estimates we used an internal proprietary tool. Note that the latency per iteration determines the clock period for the divider. We assumed 53-bit division for all the algorithms. Table 4 shows the comparison. From Table 4, Algorithms A1 and B1 offer a very small improvement of 4 percent in latency per iteration compared to a standard radix-2 SRT algorithm at the cost of 32 percent more power and 44 percent more area. Algorithms A2 and B2 offer an improvement of 10 percent in latency per iteration compared to the SRT algorithm at the cost of 36 percent more power and almost 50 percent more area. Algorithms A1, A2, B1 and B2 consume more power and area than the standard radix-2 SRT algorithm because of the following reasons: First, algorithms A1, A2, B1 and B2 must perform one of five alternatives every iteration. In comparison, the SRT algorithm must perform one of three alternatives every iteration. More alternatives directly translates to additional hardware required to update the partial remainder and quotient, which results in more power and area consumption. Second, on-the-fly conversion of a quotient digit from the set {-2, -1, 0, 1, 2} to {0, 1} is more complex than on-the-fly conversion of a quotient digit from the set {−1, 0, 1} to {0, 1}.

10 Differences

The division algorithms for the two's complement representation and the division algorithms for the BSD representations look very similar, but there are some important differences between the two. The first difference is that the range invariants for the algorithms are different. For the two's complement implementation, one range invariant for the remainder is $r \in [-2D, 2D)$. The other range invariant is $r_s \in [-2, 2)$ and $r_c \in [-2, 2)$, implying $r = r_s + r_c \in [-4, 4)$. For the BSD implementation, one range invariant for the remainder is $r \in (-2D, 2D)$. The other range invariant is $r_s \in [0, 4)$ and $r_c \in [0, 4)$, implying $r = r_c - r_s \in (-4, 4)$. Note the exclusion of both bounds in the last interval. For rounding purposes and for determining whether the computed quotient is exact, a range invariant for r that excludes both bounds is much preferable. A range that excludes the bounds can potentially save an extra clock cycle [14].

11 Concluding Remarks

In this paper we presented the derivation of four division algorithms, three of which are new. All four algorithms choose a quotient digit from the set {−2, −1, 0, 1, 2} and the selection of a quotient digit relies on only the two most-significant bits of the partial-remainder in a redundant representation. We also presented an alternative analysis method that looks at the effects of doubling and carry-save or carry-free additions in the (r_s, r_c) plane and uses invariants to prove the correctness of the division algorithms. Our method, along with the P-D diagrams, can be applied to derive higher-radix division algorithms with efficient quotient-selection functions. Because the quotient selection function for higher-radix division algorithms depend on the value of the divisor [6], we expect that there will be several (r_s, r_c) planes, each describing the quotient selection function for a particular range of the divisor.

From Table 4, the algorithms derived in this paper offer an improvement of at most 10 percent in latency per iteration which could help achieve a faster timing-closure for speed-oriented designs. The 10 percent improvement in speed comes at the cost of 50 percent more area and 36 percent more power compared to the SRT algorithm. Because a division instruction is a less frequently executed instruction, we believe that the speed of a divider has more impact on the overall system-performance than a divider's power or area consumption on the overall system's power or area. Therefore, the 10 percent improvement in speed may be worth paying for extra power and area.

Finally, the conversion of a quotient digit from a redundant digit set {−2,−1,0,1,2,} to the non-redundant digit set {0,1} can be done by means of various on-the-fly conversions [5], [6] and [3].

Acknowledgments

We thank Dr. Ivan Sutherland and Dr. Robert Daasch for their valuable comments on the previous version of this paper. We also thank our reviewers for their suggestions.

References

[1] S. Oberman and M. Flynn, "Design issues in division and other floating-point operations," *IEEE Trans. Comput.*, vol. 46, no. 2, pp. 154–161, Feb. 1997.
[2] P. Montuschi and L. Ciminiera, "Over-redundant digit sets and the design of digit-by-digit division units," *IEEE Trans. Comput.*, vol. 43, no. 3, pp. 269–277, Mar. 1994.
[3] H. Srinivas, K. Parhi, and L. Montalvo, "Radix 2 division with over-redundant quotient selection," *IEEE Trans. Comput.*, vol. 46, no. 1, pp. 85–92, Jan. 1997.

[4] N. Burgess, "A fast division algorithm for VLSI," in *Proc. IEEE Int. Conf. Comput. Des.: VLSI Comput. Processors*, Oct. 1991, pp. 560–563.
[5] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, London, U.K.: Oxford Univ. Press, 2000.
[6] M. Ercegovac and T. Lang, *Digital Arithmetic*, San Mateo, CA, USA: Morgan Kaufmann, 2003.
[7] P. Kornerup, "Digit selection for SRT division and square root," *IEEE Trans. Comput.*, vol. 54, no. 3, pp. 294–303, Mar. 2005.
[8] J. Ebergen, I. Sutherland, and A. Chakraborty, "New division algorithms by digit recurrence," in *Proc. 38th Conf. Asilomar Signals, Syst. Comput.*, vol. 2, Nov. 2004, pp. 1849–1855.
[9] N. Jamadagni and J. Ebergen, "An asynchronous divider implementation," in *Proc. 18th IEEE Int. Symp. Asynchronous Circuits Syst*, May 2012, pp. 97–104.
[10] P. Kornerup, "Digit-set conversions: Generalizations and applications," *IEEE Trans. Comput.*, vol. 43, no. 5, pp. 622–629, May 1994.
[11] D. Harris, S. Oberman, and M. Horowitz, "SRT Division architectures and implementations," in *Proc. 13th IEEE Symp. Comput. Arithmetic*, Jul. 1997, p. 18.
[12] T. N. Pham and E. E. J. Swartzlander, "Design of radix-4 SRT dividers in 65 nanometer CMOS technology," in *Proc. Int. Conf. Appl.-Specific Syst., Arch. Processors*, Sep. 2006, pp. 105–108.
[13] W. Liu and A. Nannarelli, "Power efficient division and square root unit," *IEEE Trans. Comput.*, vol. 61, no. 8, pp. 1059–1070, Aug. 2012.
[14] J. Prabhu and G. Zyner, "167 MHz radix-8 divide and square root using overlapped radix-2 stages," in *Proc. 12th Symp. Comput. Arithmetic*, Jul. 1995, pp. 155–162.
[15] M. Kuhlmann and K. K. Parhi, "A novel low-power shared division and square-root architecture using the GST algorithm," *VLSI Des.*, vol. 12, no. 3, pp. 365–376, 2001.
[16] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *IRE Trans. Electron. Comput.*, vol. EC-10, no. 3, pp. 389–400, Sep. 1961.
[17] C. Tung, "A division algorithm for signed-digit arithmetic," *IEEE Trans. Comput.*, vol. C-17, no. 9, pp. 887–889, Sep. 1968.
[18] K. D. Tocher, "Techniques of multiplication and division for automatic binary computers," *Quarterly J. Mechanics Appl. Math.*, vol. 11, no. 3, pp. 364–384, 1958.
[19] M. Ercegovac and T. Lang, "On-the-fly conversion of redundant into conventional representations," *IEEE Trans. Comput.*, vol. C-36, no. 7, pp. 895–897, Jul. 1987.

Jo Ebergen received the PhD degree in mathematics and computing science from the Eindhoven University of Technology. He is a senior consulting hardware engineer in the VLSI Research Group at Oracle Labs. His research interests include asynchronous circuit design, VLSI design, performance analysis of digital systems, and formal verification. He is a member of the IEEE and a distinguished engineer of the ACM.

Navaneeth Jamadagni received the BE degree from Visvesvaraya Technological University, Belgaum, India in 2004, and the MS degree from Portland State University, Portland, Oregon in 2010. He is a working towards the PhD degree at Portland State University. He is an intern at Oracle Labs, Redwood Shores, CA, and a graduate research assistant at the Asynchronous Research Center at Portland State University. His research interests include high-speed digital circuit design, asynchronous circuit design, and computer arithmetic. He is a student member of the IEEE Computer Society.