# Double-least-significant-bits 2′s-complement number representation scheme with bitwise complementation and symmetric range

B. Parhami

**Abstract:** A scheme is proposed for representing 2's-complement binary numbers in which there are two least-significant bits (LSBs). Benefits of the extra LSB include making the number representation range symmetric (i.e. from $-2^{k-1}$ to $2^{k-1}$ for $k$-bit integers), allowing sign change by simple bitwise logical inversion, facilitating multiprecision arithmetic and enabling the truncation of results in lieu of rounding. These advantages justify the added storage and interconnect costs stemming from the extra bit. Operation latencies show little or no change relative to conventional 2's-complement arithmetic, thus making double-LSB representation attractive.

## 1 Introduction

Novel number representation systems continue to be of much interest in computer arithmetic, with contributions appearing regularly in the main forums of the field. Even though commonly used representations (2's complement for fixed-point and ANSI/IEEE 754 standard for floating-point numbers) have achieved ubiquity as interchange and storage formats, there is still much room for innovation in the design of internal number representation formats to achieve speed, compactness and power efficiency in arithmetic circuit implementations [1]. Furthermore, certain application-specific integrated circuits and system-on-chip designs that do not need to interface with other components at the level of number representation formats can use any format that best suits the application requirements. This is where non-standard and exotic representations, such as logarithmic [2] and residue [3] number systems, find their niches.

A scheme is proposed for representing 2's-complement binary numbers in which there are two least-significant bits (LSBs). The extra LSB (ELSB) of such a double-LSB (DLSB) representation serves several purposes. It makes the number representation range symmetric, that is, from $-2^{k-1}$ to $2^{k-1}$ for $k$-bit integers instead of $-2^{k-1}$ to $2^{k-1}-1$ for ordinary 2's complement, allows numbers to be negated by simple bitwise logical inversion, facilitates multiprecision arithmetic and enables the truncation of results in lieu of rounding. Furthermore, it allows unsigned integer values in the range $[0,\ 2^k]$ to be represented faithfully, which is important in certain applications, such as when residues modulo $2^k+1$ are of interest; a standard binary format of width $k+1$ bits leads to the wider range $[0,\ 2^{k+1}-1]$, covering values that are not proper modulo-$(2^k+1)$ residues.

In a way, the DLSB representation combines the advantages of 2's-complement and 1's-complement number representation systems in terms of arithmetic efficiency. The aforementioned advantages justify the added storage and interconnect costs stemming from the extra bit. It is shown that basic arithmetic operations can be performed on DLSB numbers with little or no overhead in time and circuit complexity compared with those for ordinary 2's-complement numbers. This makes DLSB representation attractive [4]. One application area, for which DLSB representation appears to be a perfect match and does not imply any redundancy, is also pointed out.

DLSB representation constitutes a redundant number system [5, 6] that uses $k+1$ bits to represent a set of values covering roughly the same range as $k$-bit unsigned or 2's-complement system. The redundancy is at the absolute minimum of one bit, thus making the added cost trivial. With the exception of $-2^{k-1}$ and $2^{k-1}$, every value has two different representations in DLSB format. This redundancy proves beneficial in performing multiprecision arithmetic operations and in rounding steps needed in real-number arithmetic. With regard to circuit implementations, conventional design strategies based on ordinary or positively weighted bits (posibits denoted in diagrams by heavy dots '•') and negatively weighted bits (negabits denoted in our extended dot notation by small hollow circles '○') can be applied directly [7], thus allowing the use of standard, fully developed and highly optimised arithmetic circuit components, such as binary full adders, column compressors and carry acceleration structures.

The rest of this paper is organised as follows. Section 2 is devoted to the introduction of DLSB numbers and their fundamental properties, covering in particular the operations of negation and conversion to/from ordinary 2's complement. In Section 3, addition and subtraction algorithms for the DLSB representation format, including carry-save addition of several DLSB numbers are discussed. Section 4 covers the multiplication algorithms and associated hardware, using serial or parallel (array or tree) implementations. Section 5 deals with division and square rooting, implemented through digit recurrence or convergence methods. In Section 6, support operations, such as shifts and rounding, that are needed in floating-point arithmetic,

doi:10.1049/iet-cds:20070235

Paper first received 9th July and in revised form 10th October 2007

The author is with the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106-9560, USA

E-mail: parhami@ece.ucsb.edu

are presented. Section 7 is devoted to multiprecision arithmetic, using a generalised signed-digit (GSD) formulation. Conclusions and directions for further research appear in Section 8. A list of abbreviations and key notations is presented in Table 1 for ease of reference.

## 2 DLSB numbers

Consider an unsigned binary or 2's-complement number $x = (x_{k-1}x_{k-2} \dots x_1x_0 \cdot x_{-1}x_{-2} \dots x_{-l})_{\text{two}}$ and attach an ELSB to it. The resulting DLSB unsigned or 2's-complement (DLSB-2u or DLSB-2c) number is shown as $x = (x_{k-1}x_{k-2} \dots x_1x_0 \cdot x_{-1}x_{-2} \dots x_{-l}|\underline{X}_{-l})_{\text{DLSB-2u}}$ or $x = (x_{k-1}x_{k-2} \dots x_1x_0 . x_{-1}x_{-2} \dots x_{-l}|\underline{X}_{-l})_{\text{DLSB-2c}}$, with the understanding that an underlined uppercase digit, separated from the main part of the representation by a vertical bar, has the same weight as the digit to its left. For convenience, such a DLSB number is called a $(k + l)$-bit number, even though its representation really encompasses $k + l + 1$ bits.

*Example 1:* The four-bit DLSB-2c number $(10.11|\underline{0})_{\text{DLSB-2c}}$ denotes $-2 + 1/2 + 1/4 = -1.25$; the same bit string represents $2 + 1/2 + 1/4 = 2.75$ as an unsigned DLSB number.

*Example 2:* The four-bit DLSB-2u number $(11.01|\underline{1})_{\text{DLSB-2u}}$ denotes $2 + 1 + 1/4 + 1/4 = 3.5$; the same bit string corresponds to $-2 + 1 + 1/4 + 1/4 = -0.5$ as a 2's-complement DLSB number.

Except where noted, our discussion will be limited to integers; that is, $k$-bit DLSB numbers with $(k + 1)$-bit representations and the radix point at their extreme right. However, all of our results can be extended trivially to any fixed-point format.

**Table 1: List of abbreviations and key notations**

| | |
|---|---|
| • | positively weighted bit (posibit) |
| ○ | negatively weighted bit (negabit) |
| 2c | binary 2's complement |
| 2u | binary unsigned |
| DLSB | double LSB, used to refer to both the unsigned and 2's-complement versions |
| DLSB-2c | double LSB, 2's-complement |
| DLSB-2u | double LSB, radix-2 unsigned |
| ELSB | extra LSB (symbolically, ELSB is shown in uppercase and underlined; e.g. $\underline{X}_0$) |
| FA | full adder (single-bit binary adder) |
| $f$ | value of the fractional part of a number to be rounded to an integer |
| GSD | generalised signed digit |
| $h$ | shift amount |
| $k$ | number of integer positions in a fixed-point number |
| $l$ | number of fractional positions in a fixed-point number |
| LSB | least-significant bit |
| MSB | most-significant bit |
| ulp | unit in least(-significant) position; 1 for integers |
| $\varepsilon$ | signal indicating that an incoming transfer will be in [1, 2] rather than [0, 1] |
| $\sigma_{\text{AND}}$, $\sigma_{\text{OR}}$ | sticky bits used for floating-point rounding |

Fig. 1 depicts the various representations of DLSB numbers used in this paper. The ELSB can increase the numerical value of a 2's-complement number by ulp (unit in least position, 1 for integers) making the number representation range fully symmetric, that is, from $-2^{k-1}$ to $2^{k-1}$. For example, the range of four-bit signed integers, with five-bit DLSB representations, is from $-8$ to $+8$. This symmetry is the first advantage of DLSB-2c representation over the ordinary 2's-complement format. It means, for example, that negation (sign change) can never lead to overflow. Note that, in Fig. 1*c*, the leftmost negabits would change to posibits if unsigned numbers are considered.

Except for the smallest and largest representable DLSB-2c values, that is, $-2^k$ and $+2^k$ for integers, each value has two different representations: one with ELSB = 0 and the other with ELSB = 1. Thus, the DLSB number system is redundant. The redundancy includes 0, which has an all-0s and an all-1s representation in DLSB-2c. This is one disadvantage of the proposed representation and makes zero detection somewhat more difficult (similar to 1's complement). In other words, a zero detection circuit must be formed from an AND tree, to detect the all 1s pattern, and an OR tree, whose complemented output signals the all 0s pattern. Note that the representation of zero remains unique in DLSB-2u.

As in 1's complement format, the alternate representation of 0, which may be called 'minus zero', causes no problem in arithmetic operations. In other words, $+0$ and $-0$ produce identical results when combined with other numerical values via standard arithmetic algorithms. The reader can test this assertion, for example, by taking one of the operands to be $-0$ in our subsequent discussion of addition and other arithmetic algorithms.

The second advantage of DLSB representation is that the negated form of a number can be obtained by bitwise logical inversion, as justified by Theorem 1.

*Theorem 1:* The negation or 2's complement of a DLSB-2c number is obtained by simply inverting all of its $k + 1$ bits.

*Proof:* The result is proved separately for two cases. Case 1: ELSB = 0. To change the sign of a 2's-complement number, normally all the bits are inverted and ulp is added. Inverting the ELSB as part of the bitwise complementation process makes it 1, thus accomplishing this required incrementation. Case 2: ELSB = 1: The core part of the number (excluding ELSB) is one unit less than the number's true value, so its 1's complement will be one unit more than the correct 1's complement for the entire number. Because ELSB is set to 0 during bitwise inversion, the correct 2's complement of the number is obtained. □

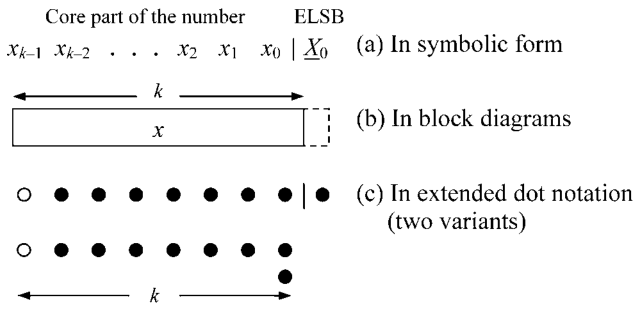

Core part of the number
ELSB
$x_{k-1}$ $x_{k-2}$ . . . $x_2$ $x_1$ $x_0$ | $\underline{X}_0$
(a) In symbolic form
$k$
$x$
(b) In block diagrams
(c) In extended dot notation (two variants)
$k$

**Fig. 1** *Representation of a k-bit DLSB number*

*Example 3:* Bitwise inversion of $(1011 \mid \underline{0})_{\text{DLSB-2c}}$, which represents $-5$, yields $(0100|\underline{1})_{\text{DLSB-2c}}$, one of the two representations of $+5$.

Whereas with standard 2's-complement representation, the addition of 1 needed to change the sign of a number causes no difficulty or extra delay when the complementation is immediately followed by a normal addition (the 1 is accommodated by setting the carry-in to 1), there are computations in which the sign change or 2's complementation does not come before an addition. It may also be the case that an addition does follow the 2's complementation, but it already has a carry-in from a previous computation step (as in multiprecision arithmetic) and thus cannot accommodate another incrementation through setting $c_{\text{in}}$ to 1.

The following interpretation of DLSB numbers is useful in designing or understanding some arithmetic algorithms.

*Theorem 2:* A DLSB-2u or DLSB-2c number can be converted to an infinitely long binary unsigned or 2's-complement number, respectively, by repeating the ELSB infinitely many times to the right of the least-significant position.

*Proof:* The repetition process converts the integer $(x_{k-1}\, x_{k-2} \ldots x_2\ x_1\ x_0|\underline{X}_0)_{\text{DLSB-2c}}$ to the infinitely wide real number $(x_{k-1}\, x_{k-2} \ldots x_2\, x_1 x_0.\ X_0\ X_0\ X_0 \ldots)_{2c}$ with a $k$-bit integral part. The equivalence of these two representations becomes clear upon noting that $\text{ulp} = \text{ulp}/2 + \text{ulp}/4 + \text{ulp}/8 + \ldots.$ □

*Example 4:* The DLSB-2c numbers $(1011 \mid \underline{0})_{\text{DLSB-2c}}$ and $(10.10 \mid \underline{1})_{\text{DLSB-2c}}$, representing the values $-5$ and $-1.25$, respectively, can be rewritten as the infinitely wide 2's-complement numbers $(1011.0000\ldots)_{2c}$ and $(10.101111\ldots)_{2c}$.

The third advantage of DLSB representations is an important one in floating-point arithmetic. When the result of a floating-point operation has more precision that the number representation format can accommodate, the extra bits must be discarded and the remaining bits adjusted through rounding. The rounding process is time consuming and adds significant delay to the critical path of most floating-point operations because, in the worst case, it requires full carry propagation. The use of an ELSB allows us to simply insert a 1 into ELSB when the rounding algorithm calls for incrementation by 1. Details will be discussed in Section 6.

## 3 Addition and subtraction

Addition of two DLSB numbers, yielding a DLSB result, is straightforward and can be done by an ordinary $k$-bit adder as shown in Fig. 2. The following discussion applies to both DLSB-2u and DLSB-2c numbers. The ELSB of one operand is used as the adder's carry-in and the ELSB of the other is attached to the output as its ELSB. Carry-out and overflow rules are exactly as in ordinary (unsigned or 2's-complement) addition. Thus, the complexity of addition is virtually unchanged relative to standard unsigned or 2's-complement numbers.

Subtraction of DLSB-2c numbers is equally simple. The subtrahend is negated, through bitwise logical inversion of its $k+1$ bits, and the result added to the minuend as above. The dashed oval box in Fig. 2 indicates where the selective complementer might be placed to allow addition or subtraction with the same $k$-bit adder. Because complementation does not require the use of the adder's carry-in,

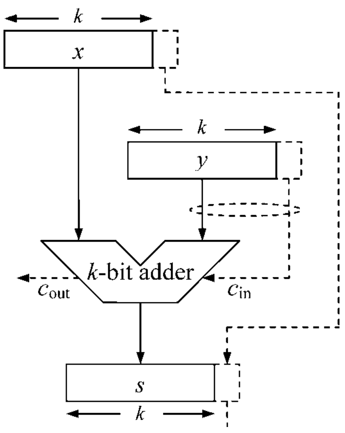


**Fig. 2** *Addition of two DLSB numbers*

a selective complementer can be used for each operand, allowing us to compute any of the functions $x+y$, $x-y$, $-x+y$ or $-x-y$ with equal ease. The last of these functions cannot be easily computed with ordinary 2's complement; computing $x+y$ and complementing the result is not as efficient since it requires an extra complementer at the output and involves carry propagation for the addition of 1 to form the 2's complement of $x+y$.

Carry-save addition of three DLSB numbers, to form two numbers of the same type, is as fast as carry-save addition with ordinary binary numbers and requires no extra hardware. Fig. 3 shows the process in dot notation. Note that for DLSB-2u inputs, all the dots in Fig. 3 will be posibits. For DLSB-2c inputs, one of the two dots (marked with an asterisk in Fig. 3) produced in the sign position is a posibit, because it is the carry-out of a full adder with three posibits as inputs. However, given that all arithmetic in 2's-complement number system is performed modulo $2^k$, replacing the posibit with a negabit will not affect the correctness of the result. This is because such a replacement reduces the value of the output vector pair by $2^{k-1} - (-2^{k-1}) = 2^k$.

If two DLSB numbers are to be added, producing a normal unsigned or 2's-complement result, that is, with ELSB = 0, then the scheme shown in Fig. 4 can be applied. Here, $k$ half-adders are used to convert the two DLSB numbers into two ordinary unsigned or 2's-complement numbers plus a carry-in for standard two-operand addition. Note that in the case of 2's-complement numbers, the leftmost carry bit (marked with an asterisk in Fig. 4) can be viewed as a posibit or negabit, as explained earlier. This addition scheme can be used just before output or communication with other digital subsystems that expect ordinary binary numbers. It

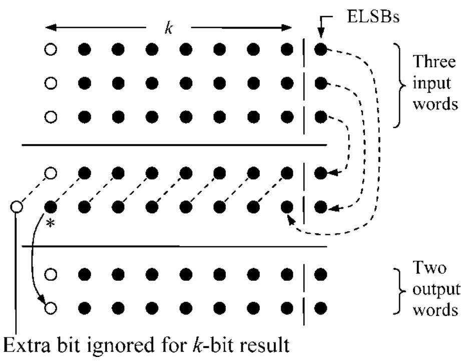


**Fig. 3** *Carry-save addition of three DLSB numbers*

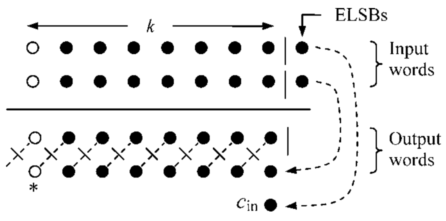


**Fig. 4** *Two DLSB numbers transformed into conventional numbers plus $c_{in}$*

can also be used as a preprocessing step when the addition of two DLSB numbers must be done with an externally supplied carry-in; this external carry-in can then be appended to the sum as its ELSB.

## 4 Multiplication

Multiplication of DLSB numbers by powers of 2 is done through left shifts, just like ordinary 2's-complement numbers, with only a slight modification.

*Theorem 3:* A DLSB number can be multiplied by $2^h$ through $h$ left shifts of the core part of the number, as is done for standard unsigned or 2's-complement numbers, except that the value of ELSB, rather than 0, is shifted in.

*Proof:* Immediate upon noting that

$$2^h(x + \underline{X}_0) = 2^h x + \underline{X}_0(2^h - 1) + \underline{X}_0$$

The first term on the right is the left-shifted $x$, the second term represents the $h$ lowest bits of the shifted number being set to $\underline{X}_0$ and the last term is the ELSB of the shifted number which has been left intact. This result can also be justified based on Theorem 2. □

Sequential multiplication of DLSB-2u or DLSB-2c numbers, using an adder of the type shown in Fig. 2, is straightforward. The only difference with unsigned or 2's-complement multiplication is that the accumulated partial product is initialised to 0 or the multiplicand $y + \underline{Y}_0$, depending on the ELSB $\underline{X}_0$ of the multiplier, instead of to 0. The reason is evident from the identity

$$(y + \underline{Y}_0)(x + \underline{X}_0) = \underline{X}_0(y + \underline{Y}_0) + \sum_{i=0}^{k-2} 2^i x_i (y + \underline{Y}_0)$$
$$\pm\, 2^{k-1} x_{k-1}(y + \underline{Y}_0)$$

The first term on the right side, $\underline{X}_0(y + \underline{Y}_0)$, is accommodated through initialisation, as stated above, and the remaining terms $x_i(y + \underline{Y}_0)$, $i = 0$ to $k - 1$, are added/subtracted in successive cycles following a right shift of the partial product. Alternatively, the extra term $\underline{X}_0(y + \underline{Y}_0)$ can be handled by an additional cycle at the beginning that is not followed by a right shift of the partial product. Note that the last term on the right is added for unsigned, and subtracted for 2's-complement, multiplication.

For $k$-bit DLSB-2u numbers, the product is in the range $[0, 2^{2k}]$, consisting of a $2k$-bit core part in $[0, 2^{2k} - 1]$ and an ELSB in [0, 1]. The above-mentioned sequential multiplication algorithm automatically obtains the product in this form and never leads to overflow. For $k$-bit DLSB-2c operands, each in the range $[-2^{k-1}, 2^{k-1}]$, the product is in the range $[-2^{2k-2}, 2^{2k-2}]$. Again, the sequential algorithm yields the correct DLSB-2c representation of the result without the possibility of overflow or need for any correction.

Tree and array multipliers can be derived based on the dot notation of Fig. 5 representing the multiplication of two four-bit DLSB-2u numbers using the identity

$$(y + \underline{Y}_0)(x + \underline{X}_0) = xy + \underline{X}_0 y + \underline{Y}_0 x + \underline{X}_0 \underline{Y}_0$$

As can be seen in Fig. 5, the height of the partial-products matrix has been increased by 2 (from 4 to 6 in this example). For some word widths $k$, this added height might result in one extra level of carry-save addition in the Wallace or Dadda tree needed to reduce the partial products to two before the final carry-propagate addition. The only exception to the maximum of one extra level occurs in the uninteresting case of $k = 3$, where reducing five partial products to two requires three levels of carry-save addition, whereas 3-to-2 reduction needs only one level.

In an array multiplier, the extra two rows of dots, depicted near the bottom of Fig. 5, can be accommodated by supplying them as inputs to the top row of cells which normally receive 0s [1]. Thus, no extra hardware is needed in the presence of such unused additive inputs. The increase in the length of the critical path, and thus the effect on latency, is negligible.

Multiplication of DLSB-2c numbers can be performed by adapting the Baugh–Wooley [8], or modified Baugh–Wooley [9], method. Only the more efficient modified Baugh–Wooley method which is based on complementing the entire AND term $x_{k-1}y_i$ or $x_i y_{k-1}$ rather than just the literal $y_i$ or $x_i$ will be considered. Fig. 6 shows the multiplication process for DLSB-2c numbers using the modified Baugh–Wooley method. The $2k - 2$ hollow circles (at the top and left side of the triangle) through which heavy dashed lines have been drawn are complemented and two 1 terms, shown in boldface, are added just to the left of the triangle. These complemented terms are then treated as posibits during the partial-products reduction process. A similar transformation is applicable to the two negabits corresponding to $\underline{X}_0 y_{k-1}$ and $\underline{Y}_0 x_{k-1}$: they are complemented, a special $-1$ term is inserted in the next higher column (to cancel the effect of replacing $\underline{X}_0 y_{k-1}$ and $\underline{Y}_0 x_{k-1}$ with their complements $1 - \underline{X}_0 y_{k-1}$ and $1 - \underline{Y}_0 x_{k-1}$, respectively), treating the complemented terms as posibits, and removing the 1 and $-1$ terms from column $k$. With these provisions, the DLSB-2c multiplication has fundamentally the same latency and complexity as DLSB-2u multiplication.

Thus far, the multiplier $(x, \underline{X}_0)$ has been used in its original form. If Booth's recoding on the multiplier is used, then the multiplication process becomes even simpler. A required result for multiplication with Booth-recoded

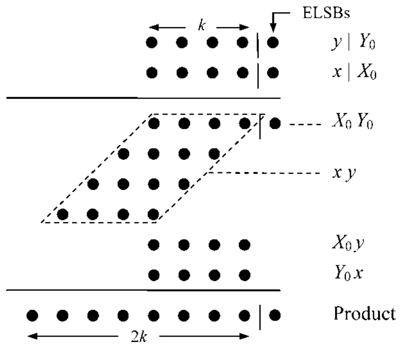


**Fig. 5** *Dot notation for 4 × 4 multiplication of DLSB-2u numbers*

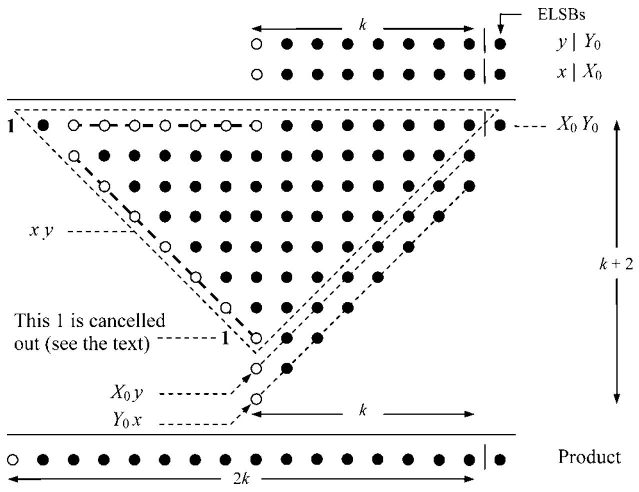


**Fig. 6** *Dot notation for $k \times k$ multiplication of DLSB-2c numbers*

multiplier is first proved. Based on this result, any 2's-complement multiplication scheme that begins with Booth's recoding is virtually unchanged for DLSB-2u or DLSB-2c numbers.

*Theorem 4:* A DLSB number can be recoded, using radix-2 or radix-4 Booth's recoding scheme, by simply considering its ELSB $\underline{X}_0$ as being $x_{-1}$ (i.e. ELSB is used as the context within which the LSB is recoded). The resulting recoded version of the multiplier will have the same width as when the number does not have an ELSB; viz. $k$ radix 2 or $\lceil k/2 \rceil$ radix-4 digits for DLSB-2c and possibly one more digit for DLSB-2u.

*Proof:* Normally, Booth's recoding of the multiplier is done by considering $x_{-1} = 0$. Based on Theorem 2, the equivalent number which has infinitely many digits of value $\underline{X}_0$ to the right of its LSB can be recoded. Given that identical digits appearing in sequence are converted to 0s as a result of recoding, setting $x_{-1}$ to the ELSB $\underline{X}_0$ yields the same result. If $\underline{X}_0 = 0$, then $x_{-1} = 0$ will not affect the recoding process. If $\underline{X}_0 = 1$, recoding with $x_{-1} = 1$ adds 1 to the core part. In both cases, the recoded multiplier has a value equal to the original multiplier $x + \underline{X}_0$. □

*Example 5:* Consider the multiplier $(1011|\underline{0})_{\text{DLSB-2c}}$, which represents $-5$. The radix-2 Booth-recoded version of this multiplier is $(\bar{1}\ 1\ 0\ \bar{1})_{\text{two}}$, and its radix-4 recoded version is $(\bar{1}\ \ \bar{1})_{\text{four}}$. Likewise, the multiplier $(1011|\underline{1})_{\text{DLSB-2c}}$, representing $-4$, becomes $(\bar{1}\ 1\ 0\ 0)_{\text{two}}$ or $(\bar{1}\ 0)_{\text{four}}$. In either case, no extra time or hardware is needed to perform the multiplication by $x + \underline{X}_0$ compared with that of an ordinary unsigned or 2's-complement multiplier $x$.

## 5 Division and square rooting

Division by powers of 2 is done through right shifts, as for normal unsigned or 2's-complement binary numbers, with only a slight modification.

*Theorem 5:* A DLSB number can be divided by $2^h$ through $h$ right shifts of its core part, as is done for ordinary unsigned or 2's-complement numbers, with the new ELSB set to the logical AND of its original value and all the $h$ bits that are shifted out.

*Proof:* Immediate from the following

$$
\begin{aligned}
\lfloor 2^{-h}(x+\underline{X}_0)\rfloor &= \lfloor 2^{-h}(2^h\lfloor 2^{-h}x\rfloor + x \bmod 2^h + \underline{X}_0)\rfloor \\
&= \lfloor 2^{-h}x\rfloor + \lfloor 2^{-h}(x \bmod 2^h + \underline{X}_0)\rfloor \\
&= \lfloor 2^{-h}x\rfloor + \lfloor 2^{-h}(x_{h-1}x_{h-2}\ldots x_1x_0|\underline{X}_0)_{\text{DLSB-2u}}\rfloor
\end{aligned}
$$

Only if the DLSB-2u number within the parentheses is equal to $2^h$, that is, all of its digits are 1s, the ELSB of the result should be set to 1. □

Digit-recurrence division [10], whether in its simple radix-2 (non-)restoring version or in its high-radix implementations, is basically an MSD-first operation. So, in principle, using the interpretation of Theorem 2, one should be able to divide DLSB numbers using essentially the same algorithms as used for unsigned or 2's-complement division. This is indeed the case.

*Example 6:* Consider dividing $34 = (100010|\underline{0})_{\text{DLSB-2u}}$ by $6 = (101|\underline{1})_{\text{DLSB-2u}}$, using the restoring binary algorithm. The quotient digits (beginning with the initial or 0th partial remainder $100010|\underline{0}$), are $q_2 = 1$ (first remainder $= 010100|\underline{0}$), $q_1 = 0$ (second remainder $= 101000|\underline{0}$) and $q_0 = 1$ (third remainder $= 100000|\underline{0}$). Thus, the quotient $(101)_{2\text{u}}$ and the final remainder $(100)_{2\text{u}}$ are obtained. It is worth noting that in a subtraction performed to obtain the trial difference, a DLSB number is subtracted from, or its bitwise complement added to, an ordinary binary number. An ordinary adder can perform this operation, without creating an ELSB at the output. This is readily understood by considering the adder of Fig. 2, with its input $x$ not having an ELSB. The ELSB of the dividend is taken into consideration only in the final cycle.

The only complication arises for basic non-restoring division, where shifting over 0s is disallowed. In this algorithm, the sign of the partial remainder determines the next quotient digit as well as the next operation to be performed (addition or subtraction), with 0 considered positive. Because 0 has two DLSB-2c representations, one with positive and the other with negative sign, direct implementation of this division algorithm may necessitate a final correction step to bring the remainder to within the allowed range. However, Sweeney–Robertson–Tocher (SRT) or high-radix division algorithms with redundant quotient digit sets work fine with little or no modification, because they rely only on a few most-significant digits of the partial remainder for selecting the next quotient digit.

Both versions of convergence division, that is, division by reciprocation [11] or by repeated multiplications [12], are also directly applicable and in fact become slightly faster in view of the fact that the 2's complementation step required to determine the next multiplicative factor is carry-free with DLSB-2c numbers. In division by repeated multiplications, for example, after an initial table lookup to determine an approximate reciprocal of the divisor $d$, both the dividend $z$ and the divisor $d$ are multiplied by a sequence of multipliers, the first of which is the approximate reciprocal and each subsequent one is the 2's complement of a previous result. As discussed in Section 4, the required multiplications may become slightly more complicated with DLSB-2c numbers. However, the removal of the addition step needed for 2's complementation with ordinary 2's-complement numbers more than makes up for the loss in multiplication speed.

As in convergence division with ordinary binary operands, the two multiplications needed for each iteration can be pipelined. In this case, the removal of the addition step for 2's complementation helps keep the pipeline full at all times, leading to even better performance.

Considerations for square rooting, both through digit recurrence and by convergence, are quite similar to those for division. Again the simpler 2's complementation process (required for the second equation below) helps improve the performance of the following square-rooting algorithm for computing $\sqrt{z}$ based on the Newton–Raphson method

$$x^{(i+1)} = 0.5(x^{(i)} + zy^{(i)})$$
$$y^{(i+1)} = y^{(i)}(2 - x^{(i)}y^{(i)})$$

In the convergence method above, which requires three multiplications per iteration, $y$ will tend to $1/\sqrt{z}$ and $x$ to $\sqrt{z}$.

## 6 Floating-point arithmetic

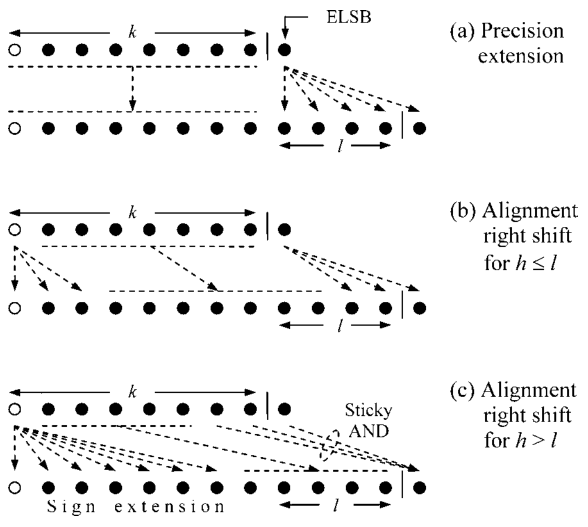
Our discussion of floating-point arithmetic with DLSB numbers must include conversion of a DLSB significand to an internal extended-precision representation, alignment preshift (for addition or subtraction), normalisation postshift and rounding [1].

Suppose that a DLSB integer $(x_{k-1}x_{k-2}\ldots x_1x_0|\underline{X}_0)_{\text{DLSB}}$ must be extended to maintain its $k$ integer bits and additionally to have $l$ fractional bits internally. This is easily accomplished by replicating the ELSB in the $l$ fractional bits as well as in the new ELSB at position $-l$. That is

$$(x_{k-1}x_{k-2}\ldots x_1x_0|\underline{X}_0)_{\text{DLSB}} = (x_{k-1}x_{k-2}\ldots x_1x_0\,.X_0X_0\ldots X_0|\underline{X}_0)_{\text{DLSB}}$$

The transformation above, which is easily deduced from Theorem 2, is represented graphically in Fig. 7*a*, using our extended dot notation. The dashed arrows represent direct copying of bits or blocks of bits.

For alignment preshift by $h$ bits, two cases are distinguished. For $h \le l$, where $l$ is the precision increase for the internal representation, shifting is done with sign extension at the left end and ELSB extension at the right. This is depicted in Fig. 7*b* and can be easily accomplished by precision extension followed by ordinary 2's-complement shifting. For $h > l$, 2's-complement shifting is done with the provision that all the bits shifted off the right end are ANDed together to yield the value for the ELSB (Fig. 7*c*). This operation is called 'sticky AND' in analogy with the OR operation performed for the 'sticky' bit in ordinary floating-point arithmetic. The latter bit is called the 'sticky OR' bit for clarity. Note that if the alignment right shift is preceded by precision extension and has the provision for sticky AND on the bits that are shifted out from the right end, the correct operation is performed in both cases $h \le l$ and $h > l$.



k
ELSB
(a) Precision extension
l
k
(b) Alignment right shift for h ≤ l
l
k
Sticky AND
(c) Alignment right shift for h > l
Sign extension
l

**Fig. 7** *Graphical representation of precision extension and alignment preshift for DLSB numbers*

Normalisation shifts are essentially multiplication and division by powers of 2 which are already discussed in Sections 4 and 5 (Theorems 3 and 5). The only additional consideration is that, in the case of right shifts, the sticky AND is performed past bit position $-l$ rather than past position 0, as specified in the statement of Theorem 5.

Next, the round-to-nearest-even operation is discussed in detail. Other rounding modes are similar. Assume that an extended-precision DLSB operand $(x_{k-1}x_{k-2}\ldots x_1x_0.x_{-1}x_{-2}\ldots x_{-l}|\underline{X}_{-l})_{\text{DLSB}}$ is to be rounded to an integer $(x_{k-1}x_{k-2}\ldots x_1x_0|\underline{X}_0)_{\text{DLSB}}$. Note that the rounded value has exactly the same bits as the interim result in positions $k-1$ down to 0; the only thing that needs to be determined for proper rounding is the value of the ELSB $\underline{X}_0$ in the rounded result. Thus, rounding of DLSB numbers can be considerably faster than ordinary rounding performed for unsigned or 2's-complement binary numbers.

Assuming that the fractional part $(.x_{-1}x_{-2}\ldots x_{-l}|\underline{X}_{-l})_{\text{DLSB}}$ of the extended-precision interim result has the value $f$ in [0, 1], the round-to-nearest-even rounding scheme can be defined as follows

$$\text{Round down if } f < \frac{1}{2} \text{ or if } f = \frac{1}{2} \text{ and } x_0 = 0$$
$$\text{Round up if } f > \frac{1}{2} \text{ or if } f = \frac{1}{2} \text{ and } x_0 = 1$$

Note that, because of the ELSB, the 'fractional' part $f$ can equal 1, but the rules just given handle this case properly as well.

Rounding down is done by simply chopping the fractional part, which is always positive, and setting $\underline{X}_0$ to 0. Rounding up is accomplished by discarding the fractional part and setting $\underline{X}_0$ to 1. All that remains is to show how the three conditions $f < 1/2$, $f > 1/2$ and $f = 1/2$ are tested.

Let us designate $x_{-1}$ as the 'round bit' and define the sticky OR and sticky AND bits as follows

$$\sigma_{\text{OR}} = 0 \text{ iff } x_{-2} = x_{-3} = \cdots = x_{-l} = \underline{X}_{-l} = 0$$
$$\sigma_{\text{AND}} = 1 \text{ iff } x_{-2} = x_{-3} = \cdots = x_{-l} = \underline{X}_{-l} = 1$$

These sticky bits can be formed based on the bits $x_{-2}$ through $\underline{X}_{-l}$ of the interim result when needed. In many cases, however, only the two sticky bit values, $\sigma_{\text{OR}}$ and $\sigma_{\text{AND}}$, need to be kept in lieu of these $l$ bits. With round and sticky bits as defined above, the conditions become

$$f < \frac{1}{2} \text{ iff } x_{-1} = 0 \text{ and } \sigma_{\text{AND}} = 0$$
$$f > \frac{1}{2} \text{ iff } x_{-1} = 1 \text{ and } \sigma_{\text{OR}} = 1$$
$$f = \frac{1}{2} \text{ otherwise}$$

The ability to do rounding with no carry propagation, at the cost of maintaining an extra sticky bit (which is not needed with conventional binary numbers), is an important advantage of DLSB number representation.

The advantages above are not limited to bit-parallel arithmetic. In a companion paper [13], the author has shown how the use of DLSB representation solves one of the problems in pipelined, digit–serial arithmetic. In on-line arithmetic [14], redundant operands enter arithmetic units one digit at a time, beginning from their most significant ends, with the result digits also produced one at a time, after a short operation-dependent latency. Thus, if at the end of the output generation, it is discovered that the output should be rounded up (based on the residual that is observed), the change cannot be incorporated in the output, which has already moved downstream in the pipeline and partially processed by other units. The use of ELSB remedies this problem, as it allows us to properly increment the output, in order to produce its correctly rounded value, by simply attaching an appropriate ELSB value at the least-significant end of it.

## 7 Multiprecision arithmetic

Consider the representation of large unsigned integers in radix $2^k$, with each radix-$2^k$ digit being a DLSB-2u number. The radix-$2^k$ digit set used is thus $[0,\ 2^k]$. The result is an unsigned-digit redundant representation system with the redundancy index $\rho = 1$; it uses an extra digit value, $2^k$, compared with the conventional radix-$2^k$ representation based on the digit set $[0,\ 2^k - 1]$. As shown in Fig. 8, the resulting $g$-word representation can be viewed as a $gk$-position hybrid-redundant number [15, 16]; most digits are binary digits in [0, 1], with every $k$th digit being redundant with a value in [0, 2]. In the rest of this section, arithmetic algorithms on such hybrid redundant numbers are briefly discussed.

Consider now the addition of two such multiprecision numbers $(x^{(g-1)}, x^{(g-2)}, \ldots, x^{(0)})$ and $y = (y^{(g-1)}, y^{(g-2)}, \ldots, y^{(0)})$. For simplicity, it is assumed that the two numbers are composed of the same number $g$ of words, but this does not have to be the case. According to the rules of GSD addition for this radix-$2^k$ stored-carry number system [6], the transfer digit from one segment to the next higher segment is in [0, 2]. To ensure that transfers are absorbed and further propagation is avoided, each segment must anticipate whether its incoming transfer digit value will be in [0, 1], the lower subrange or in [1, 2], the upper subrange. The required anticipation logic consists of a single two-input AND gate, because the transfer into segment $i$ can exceed 1 only when $x^{(i)}_{k-1} = y^{(i)}_{k-1} = 1$. To see that a transfer digit of 2 is impossible when $x^{(i)}_{k-1} + y^{(i)}_{k-1} \leq 1$, assume, without loss of generality, that $x^{(i)}_{k-1} = 0$ and $y^{(i)}_{k-1} = 1$. Then, the value of $x^{(i)}$ is at most $2^{k-1}$ (which occurs when all other bits including the ELSB are 1s), whereas the value of $y^{(i)}$ does not exceed $2^k$. Thus, the addition $x^{(i)} + y^{(i)}$ leads to a maximum sum of $2^k + 2^{k-1}$, which implies a carry of at most 1 into the next segment.

It is now easy to see that parallel processing can be readily applied to the addition of $g$-segment numbers. For each segment $i$, a position sum $x^{(i)} + y^{(i)} + \varepsilon_i$ in $[0,\ 2^{k+1} + 1]$ is computed, where $\varepsilon_i$ is the anticipation signal for the transfer into segment $i$; $\varepsilon_i = 1$ means that the incoming transfer will be in [1, 2]. The position sum is a $(k+2)$-bit binary number $p^{(i)}_{k+1}p^{(i)}_k p^{(i)}_{k-1} \ldots p^{(i)}_1 p^{(i)}_0$, where $p^{(i)}_{k+1} = 1$, occurring very rarely, indicates a transfer value of 2. Because one unit of an

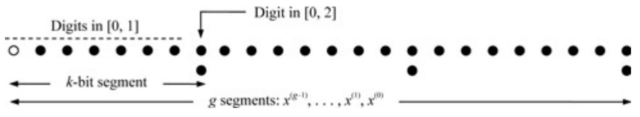


**Fig. 8** *Multiprecision DLSB representation interpreted as a hybrid-redundant number system*

**Table 2: Speed and cost penalties of DLSB over 2's complement for integer arithmetic**

| Arithmetic circuit | Extra latency | Added complexity |
|---|---|---|
| adder | 0 | 0 |
| multioperand adder | 0 | 0 |
| subtractor | 0[a] | 0 |
| serial multiplier | $\leq$1 cycle | negligible |
| Booth multiplier | 0 | 0 |
| tree multiplier | $\leq$FA delay | $\leq 2k$ FA cost |
| array multiplier | negligible | 0 |
| serial divider | $\leq$1 cycle | negligible |
| convergence divider | 0[a] | same as multiplier |
| serial square rooter | $\leq$1 cycle | negligible |
| convergence square rooter | 0[a] | same as multiplier |

[a]In this case, not only there is no speed penalty but the circuit may be faster

incoming transfer in the upper subrange [1, 2] has already been accommodated in the position sum via adding $\varepsilon_{i+1}$, the remainder of it, which is in [0, 1], can be attached as the new ELSB, just like an incoming transfer in the lower subrange [0, 1]. Thus, the addition process is completed by taking $p^{(i)}_{k-1} \ldots p^{(i)}_1 p^{(i)}_0$ as the main part and $\varepsilon_i \oplus p^{(i-1)}_k$ as the ELSB. Note that for $\varepsilon_i = 0$, $p^{(i-1)}_k$ is stored directly as the ELSB, whereas for $\varepsilon_i = 1$, the complement of $p^{(i-1)}_k$ (which is the same as one less that the two-bit binary number $p^{(i-1)}_{k+1} p^{(i-1)}_k$ is stored). This represents a simple variation on the stored-transfer number representation scheme, whereby a transfer digit from a position $i$ to the next higher position $i+1$ is not combined with the interim digit value in that position, but is rather saved alongside with it [17].

Signed multiprecision numbers can be represented and processed in a number of ways. One way is to use 2's-complement representation for the $gk$-digit number as a whole. A second way is to use DLSB-2c numbers as the $k$-bit segments, leading to radix-$2^k$ GSD representation with the signed digit set $[-2^{k-1},\ 2^{k-1}]$. The transfers will now be in $[-1,\ 1]$ but parallel processing in addition is still feasible with suitably modified rules. The latter scheme directly corresponds to hybrid signed-digit representations as proposed in [15].

## 8 Conclusion

To summarise, DLSB-2c number representation has at least four advantages compared with the ordinary 2's-complement representations.

1. Symmetric number range (no overflow on negation).
2. Negation by bitwise inversion (carry-free).
3. Rounding by truncation and setting of ELSB (carry-free).
4. Simple parallel multiprecision arithmetic (carry-free).

Given that basic arithmetic operations on DLSB-2c numbers are at most marginally slower or more complex than those for ordinary 2's-complement numbers, the resulting overhead in circuit complexity and the one-bit redundancy in number representation may be justifiable by the advantages listed above.

Table 2 summarises the comparative claims made in the preceding pages with regard to extra latency and added circuit complexity for integer arithmetic operations as one moves from ordinary 2's complement to DLSB representation. The entries for tree and array multipliers in Table 2 need some elaboration. For tree multiplier, the partial-products reduction tree will have to include two additional $k$-bit carry-save adders, with its depth potentially increasing by one level. For array multiplier, lack of a cost penalty is because of the use of additive inputs to accommodate the extra two rows of bits. Such additive inputs are present in nearly all modern array multiplier designs in order to render the design of cells more uniform for VLSI realisation. In the case of convergence division, the comparison depends on the type of multiplier used (it is almost always a tree multiplier). Floating-point operations have more favourable comparisons in view of the faster and less complex rounding operation with DLSB representation. Even though detailed implementation studies are needed to determine under what conditions DLSB numbers might be beneficial, addition of this technique to the repertoire of arithmetic system designers does provide them with new design options and trade-off tools.

There is one particular application for DLSB representation where the ELSB can be viewed as an absolute requirement, rather than as a liability. This application arises in residue number system representations with a modulus of the form $2^k + 1$. Examples of such RNS representations abound, and only one recent contribution is cited as a representative example [18]. The residues for such a modulus belong to the range $[0, 2^k]$, thus requiring at least $k + 1$ bits in any radix-2 representation. Using the DLSB format in this context [19] presents definite advantages over another well-studied and widely used representation in which a particular code is reserved for 0 and each of the $2^k$ non-zero residues is represented in a 'diminished-one' format, that is, as one unit less than its true value. Whereas the latter scheme, first proposed by Leibowitz [20] and subsequently developed and used by many others, has been shown to be preferable to straight binary representation, it does lead to some complications because of the need for detecting special cases and making adjustments in the results of arithmetic operations. Using the DLSB format not only removes these problems, but also allows the use of standard arithmetic components.

One drawback of the proposed representation is that it yields two codes for 0. However, this is only a minor inconvenience in zero detection and does not affect the validity of the arithmetic algorithms all of which properly deal with $-0$ as 0. In fact, there is a positive side to this dual representation. The round-to-nearest-even algorithm, discussed at the end of Section 6, rounds down positive values in $[0, 0.5]$ to $+0$ and rounds up negative values in $[-0.5, 0]$ to $-0$. Thus, the sign of zero can be viewed as carrying useful information in these cases. Of course, when a unique representation of 0 is prescribed, as in ANSI/IEEE standard format, automatic conversion from $-0$ to $+0$ must take place.

Other directions for further research include logic-level and circuit-level comparisons based on actual implementations to assess the cost-effectiveness of the proposed approach in greater detail than that shown in Table 2. Extension to other algorithms and implementation strategies (such as the use of additive multiply modules for multiplication) might also be attempted.

## 9 Acknowledgment

A preliminary version of this paper was published in the Proceedings of the International Conference on Signal and Image Processing [4]. The author is indebted to Stephan Johansson for his assistance during the early development of these ideas.

## 10 References

1 Parhami, B.: 'Computer arithmetic: algorithms and hardware designs' (Oxford University Press, New York, 2000)

2 Coleman, J.N., Chester, E.I., Softley, C.I., and Kadlec, J.: 'Arithmetic on the European logarithmic microprocessor', *IEEE Trans. Comput.*, 2000, **49**, pp. 702–715

3 Omondi, A., and Premkumar, B.: 'Residue number systems: theory and implementation' (Imperial College Press, London, 2007)

4 Parhami, B., and Johansson, S.: 'A number representation scheme with carry-free rounding for floating-point signal processing applications'. Proc. Int. Conf. Signal and Image Processing, 1998, pp. 90–92

5 Avizienis, A.: 'Signed-digit number representation for fast parallel arithmetic', *IRE Trans. Electron. Comput.*, 1961, **10**, pp. 389–400

6 Parhami, B.: 'Generalized signed-digit number systems: a unifying framework for redundant number representations', *IEEE Trans. Comput.*, 1990, **39**, pp. 89–98

7 Jaberipur, G., and Parhami, B.: 'Stored-transfer representations with weighted digit-set encodings for ultrahigh-speed arithmetic', *IET Circuits Devices Syst.*, 2007, **1**, pp. 102–110

8 Baugh, C.R., and Wooley, B.A.: 'A two's complement parallel array multiplication algorithm', *IEEE Trans. Comput.*, 1973, **22**, pp. 1045–1047

9 Dadda, L.: 'Fast multipliers for two's complement numbers in serial form'. Proc. 7th IEEE Symp. Computer Arithmetic, 1985, pp. 57–63

10 Ercegovac, M.D., and Lang, T.: 'Division and square root: digit-recurrence algorithms and implementations' (Kluwer, Boston, 1994)

11 Ferrari, D.: 'A division method using a parallel multiplier', *IEEE Trans. Comput.*, 1967, **16**, pp. 224–226

12 Flynn, M.J.: 'On division by functional iteration', *IEEE Trans. Comput.*, 1970, **19**, pp. 702–706

13 Parhami, B.: 'On producing exactly rounded results in digit–serial on-line arithmetic'. Proc. 34th Asilomar Conf. Signals, Systems, and Computers, 2000, pp. 889–893

14 Ercegovac, M.D., and Lang, T.: 'On-line arithmetic: a design methodology and applications'. Proc. IEEE Workshop on VLSI Signal Processing, 1988, pp. 252–263

15 Phatak, D.S., and Koren, I.: 'Hybrid signed-digit number systems: a unified framework for redundant number representations with bounded carry propagation chains', *IEEE Trans. Comput.*, 1994, **43**, pp. 880–891

16 Phatak, D.S., Goff, T., and Koren, I.: 'Constant-time addition and simultaneous format conversion based on redundant binary representations', *IEEE Trans. Comput.*, 2001, **50**, pp. 1267–1278

17 Jaberipur, G., Parhami, B., and Ghodsi, M.: 'A class of stored-transfer representations for redundant number systems'. Proc. 35th Asilomar Conf. Signals, Systems, and Computers, 2001, pp. 1304–1308

18 Cao, B., Chang, C.-H., and Srikanthan, T.: 'A residue-to-binary converter for a new five-moduli set', *IEEE Trans. Circuits Syst. I*, 2007, **54**, pp. 1041–1049

19 Jaberipur, G.: 'A one-step modulo $2^n + 1$ adder based on double-LSB representation of residues', *CSI J. Comput. Sci. Eng.*, to appear in 2008

20 Leibowitz, L.M.: 'A simplified binary arithmetic for the Fermat number transform', *IEEE Trans. Acoust. Speech Signal Process.*, 1976, **24**, pp. 356–359