On Producing Exactly Rounded Results in Digit-Serial on-Line Arithmetic

Behrooz Parhami

Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA
E-mail: parhami@ece.ucsb.edu

Abstract

The input and output of an on-line computation involving redundant numbers must generally be compatible with standard nonredundant formats. When rounding of a result is required in the last computation step, the process can be combined with on-the-fly conversion of the redundant output into nonredundant form and thus introduces no extra delay. However, if multiple on-line operations are to be performed before obtaining a final result, the preceding method would be inapplicable to the intermediate results. We present a solution scheme where rounding (subtraction of ulp, no action, addition of ulp) is represented by an additional digit attached to the number, thus obviating the need for modifying any of the previously produced digits. This added digit forms a second least-significant digit (LSD), leading to the designation "double-LSD numbers" for the resulting representation. We show that on-line arithmetic with double-LSD numbers is only slightly more complex than with ordinary signed-digit numbers.

Keywords: ASIC design, digit-pipelined arithmetic, double-LSD representation, DSP hardware, MSD-first computation, on-line algorithm, redundant number system.

1. Introduction

On-line arithmetic with signed-digit and other redundant number formats [Parh90] is an attractive alternative to conventional representations in some signal processing applications [Irwi87], [Erce88]. Because input and output must generally be compatible with standard nonredundant formats, practical systems incorporate binary-to-redundant and redundant-to-binary converters at the input and output interfaces. The former converter is often trivial and the latter one can be designed to derive the nonredundant form of the result on the fly, thus avoiding carry propagation altogether [Erce87]. Typically, on-line computations leave a residual which must be used to round the final result in a manner compatible with the rounding method in use (e.g., IEEE floating-point standard requirements). A naive implementation will introduce additional delay for rounding, but again it is possible to incorporate the required modifications for rounding in the on-the-fly conversion process [Erce92], thus avoiding extra delay at the expense of some added circuitry.

The process outlined above works fine for rounding a final result which must be converted to nonredundant form. However, if multiple on-line operations are required before obtaining a final result, the preceding method cannot be applied to the rounding of intermediate results which are not converted to nonredundant format. In a digit-parallel system, the addition or subtraction of *ulp* (unit in least-significant position) needed to effect rounding, is easily done by a carry-free addition step. However, because this increment/decrement operation may affect many of the result digits, it is unsuitable for digit-serial on-line mode of computation in which most higher-order digits have already been produced and passed on to subsequent function units.

In this paper, we present a solution scheme in which the rounding operation (subtract *ulp*, no action, add *ulp*) is represented by an additional digit attached to the number, thus obviating the need for modifying any previously produced digit. This added digit essentially forms a second least-significant digit (LSD), leading to the designation "double-LSD numbers" for the resulting representation. This is the signed-digit counterpart of double-LSB format, previously proposed by this author for standard binary numbers with signed-magnitude or two's-complement representation [Parh98]. For the proposed double-LSD format to be useful, we must show that on-line arithmetic operations can be performed on such numbers with reasonable efficiency; i.e., little added overhead compared to ordinary (non)redundant numbers.

The bulk of this paper assumes binary signed-digit (BSD) numbers with radix $r = 2$ and digit set $\{-1, 0, 1\}$. Notationally, we represent a double-LSD BSD number by using $\dot{+}$ for the digit value -1 and following the original LSD with a backslash and the second LSD. For example, $0.1\dot{+}01\backslash 1$ is one possible representation for the binary fraction $x = 0.0110$ $(1/2 - 1/4 + 0/8 + 1/16 + 1/16$, where the final 1/16 term is the contribution of the second LSD). In Section 2, we offer a brief overview of the rounding problem. Section 3 is devoted to algorithms for fixed-point arithmetic on double-LSD BSD numbers, with needed modifications for floating-point arithmetic following in Section 4. Extensions to other radices and digit sets appear in Section 5. Section 6 contains our conclusions.

0-7803-6514-3/00/$10.00©2000 IEEE

2. Rounding and Double-LSD Numbers

Consider BSD fractions of the form $x = .\, x_{-1}x_{-2} \ldots x_{-k}$. Any two consecutive numbers representable in this k-digit fixed-point format differ by *ulp*, which in this case is 2^{-k}. The process of converting a more precise number, given as input or produced in some computation step, to the foregoing format is called "rounding". In the simplest rounding scheme, known as truncation or chopping, extra digits beyond the kth position are dropped, thus leading to a maximum error of about one *ulp*. When used with ordinary binary numbers, the average error introduced by chopping is also of some concern; but this is not an issue with BSD numbers, given the symmetric digit set.

Commonly used rounding methods, on the other hand, often limit the maximum error to *ulp*/2 (which is the best possible) and have a near-zero average error due to positive and negative errors canceling each other out in the long run. Again, average error is not an issue for BSD numbers. To achieve the maximum error goal of *ulp*/2 (rounding to a nearest representable value) the dropping of extra digits may have to be accompanied by an adjustment in the amount of *ulp* to the rest of the number; adding *ulp* when the dropped part of the number is > *ulp*/2 subtracting *ulp* when it is < –*ulp*/2, with special rules applying to the case of exact equality with ±*ulp*/2. Note that the negative values just mentioned do not arise in standard binary but must be considered in BSD arithmetic.

Besides round-to-nearest methods discussed in the preceding paragraph, we sometimes need directional rounding schemes which guarantee the error to be in a desired direction. Both upward- and downward-directed rounding necessarily have a maximum error of about one *ulp*. Again, the required adjustment to the first k digits may involve adding or subtracting *ulp*. Rounding numbers so as to satisfy the strict requirements of a given rounding scheme is known as exact or proper rounding.

For BSD representation, the aforementioned adjustments may be in either direction, leading to the requirement for adding ±*ulp* to the first k digits. This addition, which is quite simple and fast in digit-parallel BSD arithmetic, would be impossible with MSD-first digit-pipelined operation, in view of more significant digits already having been passed on to, and used by, the next computational element. The double-LSD representation provides just the mechanism needed to solve this problem. When the decision is made as to how the first k digits must be adjusted for proper rounding (subtract *ulp*, no change, or add *ulp*), this adjustment is tacked on to the result as a second LSD, which is itself a digit in {–1, 0, 1}. The rest of this paper, then deals with how arithmetic is to be performed on operands that possess two LSDs due to the rounding of intermediate results.

3. Fixed-Point Fractional Arithmetic

We first discuss arithmetic operations on two double-LSD BSD numbers $x = .\, x_{-1}x_{-2} \ldots x_{-k}\backslash a$ and $y = .\, y_{-1}y_{-2} \ldots y_{-k}\backslash b$, where a and b represent the second LSDs. The following mathematical result provides insight into the double-LSD BSD number format and leads to simple modified circuits for arithmetic on such numbers in some cases.

Theorem 1: Given the double-LSD BSD representation of x as $.\, x_{-1}x_{-2} \ldots x_{-k}\backslash a$, we can represent x by the infinite string of digits $x = .\, x_{-1}x_{-2} \ldots x_{-k}\, a\, a\, a \ldots$, where the BSD digit a is repeated ad infinitum.

3.1. Addition and Subtraction

Digit-serial MSD-first addition/subtraction is performed with no modification other than allowing one more cycle to process the extra LSDs. This is easily justified based on Theorem 1. Consider $x + y$ as an example, where:

$$x = .1\bar{1}01\backslash 1 = .1\bar{1}01\ 1111\ 1111 \ldots$$
$$y = .01\bar{1}0\backslash 1 = .01\bar{1}0\ 1111\ 1111 \ldots$$

In the cycles when the last two digits $z_{-(k-1)}$ and z_{-k} of the sum are produced, x and y must be extended to the right with their second LSDs, rather than with 0s. The same is done in the extra cycle for producing c, the second LSD of the sum. Note that from this point on, any additional cycle would generate the same sum digit; this is consistent with the interpretation offered by Theorem 1.

For completeness, we mold double-LSD BSD add/subtract into a recurrence on the scaled residual $W^{(i)}$ [Erce88]. Setting $W^{(0)} = z_{-1} = 0$, we have the main recurrence:

$$W^{(i)} = 2W^{(i-1)} + x_{-i} \pm y_{-i} - 4z_{-(i-2)} \triangleq V^{(i)} - 4z_{-(i-2)}$$

With $W^{(i-1)}$ in [–2, 2], $V^{(i)}$ evaluates to an integer in [–6, 6]; for $V^{(i)}$ in [–6, –2], [–2, 2], [2, 6], the sum digit $z_{-(i-2)}$ may be set to –1, 0, 1, respectively. Overlaps between the selection intervals can be used for hardware optimization. Sum digit production begins with z_0, which forms the transfer-out of our fractional addition. If overflow is not to occur, we must be able to set $z_0 = 0$, implying that $V^{(2)}$ must be in [–2, 2]. After $z_{-(k-2)}$ has been produced, $W^{(k)}$ should not be left-shifted in the next cycle, because the extra LSDs are of the same weight as the just-processed LSDs. But for the sake of uniformity, we might continue with

$$W^{(k+1)} = 2W^{(k)} + 2a \pm 2b - 4z_{-(k-1)} \triangleq V^{(k+1)} - 4z_{-(k-1)}$$
$$W^{(k+2)} = 2W^{(k+1)} - 4z_{-k} \triangleq V^{(k+2)} - 4z_{-k}$$
$$W^{(k+3)} = 2W^{(k+2)} - 8c \triangleq V^{(k+3)} - 8c$$

where $V^{(k+1)}$ ($V^{(k+2)}$, $V^{(k+3)}$) is in [–8, 8] and divisible by 2 (4, 8, respectively); this allows the residual to be zeroed out.

3.2. Multiplication

Before discussing how the extra LSDs might be taken into consideration, a review of on-line BSD multiplication $z = x \times y$ is in order. A scaled residual is used to select the product digits with three cycles of delay relative to the inputs. Assuming that $x^{(i)}$ and $y^{(i)}$, both in [0, 1), represent the magnitudes of inputs x and y up to their $-i$th digits, the main recurrence, with $W^{(0)} = z_2 = z_1 = z_0 = 0$, is:

$$W^{(i)} = [2W^{(i-1)} + x_{-i}y^{(i)} + y_{-i}x^{(i-1)}] - 8z_{-(i-3)} \cong V^{(i)} - 8z_{-(i-3)}$$

The real-valued residual $W^{(i)}$ is in $[-6 - 2^{-i}, 6 + 2^{-i}]$. Proper choice of $z_{-(i-3)}$ allows us to maintain this condition for all i, given the fact that $V^{(i)}$, incorporating $2W^{(i-1)}$ along with terms of maximum magnitudes $1 - 2^{-i}$ and $1 - 2^{-(i-1)}$, will be confined to $[-14 - 2^{-i}, 14 + 2^{-i}]$. Once we have $z_{-(k-3)}$ at hand, leaving the residual $W^{(k)}$ in $[-6 - 2^{-k}, 6 + 2^{-k}]$, the last three digits of the product are obtained via the same recurrence, with new x and y digits set to 0. The final residual $W^{(k+3)}$ will hold the rounding information. This information can be used by producing a "round digit" $z_{-(k+1)}$ in an extra cycle and basing the rounding action on this extra digit and the sign of $W^{(k+4)}$ (a type of "sticky digit").

$z_{-(k+1)}$	$W^{(k+4)}$	Action
–1	negative	Subtract *ulp*
–1	zero	Apply midway rule
–1	positive	None
0	don't care	None
1	negative	None
1	zero	Apply midway rule
1	positive	Add *ulp*

It might appear that extending x and y with their second LSDs, as was done for addition, would allow us to multiply double-LSD BSD numbers with the same recurrence used for BSD multiplication. Unfortunately, this is not possible if properly rounded results are to be obtained. The following example shows why. Assume that

$$x = 2^{-2} + 2^{-(k-1)} \quad \text{and} \quad y = 2^{-2}$$

are k-digit BSD fractions and let $x^+ = x + 2^{-k}$, $x^- = x - 2^{-k}$, $y^+ = y + 2^{-k}$, and $y^- = y - 2^{-k}$; these are, in effect, x and y with a second LSD of 1 or –1, respectively. We have:

$$z = x \times y = 2^{-4} + 2^{-(k+1)}$$
$$z' = x^+ \times y^- = x \times y - 3 \times 2^{-2k}$$
$$z'' = x^- \times y^+ = x \times y + 2^{-2k}$$

We note that z has a round digit of 1, as do z' and z''. The latter two numbers must be rounded in opposite directions, but the required information for this action won't become available until much later; furthermore, the residual must be kept with double the normal precision.

A direct algorithm, similar to that used for addition can be applied here. The algorithm proceeds as in normal BSD multiplication until $z_{-(k-3)}$ has been produced, yielding the residual $W^{(k)}$. The extra LSDs, to be examined next, have the same weight as the regular LSDs just processed. Thus, the residual is not left-shifted (or, alternatively, other terms of $V^{(k+1)}$ in square brackets are left-shifted), yielding:

$$W^{(k+1)} = [2W^{(k)} + 2ay^{(k)} + 2bx^{(k)} + 2^{-k+1}ab] - 8z_{-(k-2)}$$

Four more iterations, with a and b digits set to 0 in the preceding equation, produce the final digits $z_{-(k-1)}$ and z_{-k}, the round digits $z_{-(k+1)}$ and $z_{-(k+2)}$, and the final residual $W^{(k+5)}$. The reason one round digit is inadequate can be explained as follows. Beginning with $W^{(k+1)}$, the scaled residuals have a wider range of about (–8, 8), which is double the range that we would have had with no extra LSD. Producing an additional round digit $z_{-(k+2)}$ ensures that the final (true) residual will not exceed *ulp*/2.

Intuitively, the reason that the preceding algorithm succeeds where our first attempt failed is that the contributions of the second LSDs to the product is taken into account at once rather then in pieces. Thus, a true final residual is at hand that allows us to round the result properly in every case. Rounding rules are as follows:

$2x_{-(k+1)}+x_{-(k+2)}$	$W^{(k+5)}$	Action
–3	don't care	Subtract *ulp*
–2	negative	Subtract *ulp*
–2	zero	Apply midway rule
–2	positive	None
–1, 0, 1	don't care	None
2	negative	None
2	zero	Apply midway rule
2	positive	Add *ulp*
3	don't care	Add *ulp*

In the following 4-digit multiplication example, the true, rather than the scaled or left-shifted, residual is shown to make the process more intuitive and thus clearer.

```
  .1ī01\1
× .10ī1\ī
  ---------
  .01          residual at the end of cycle 1, W(1)/2
  .01ī0        add terms in cycle 2 to get W(2)/4
  .01īī 10     add terms in cycle 3
  .01īī 10     p(1) = .0          no adjustment, W(3)/8
  .01ī0 1ī11   add terms in cycle 4
  .00ī0 1ī11   p(2) = .01         adjust residual, W(4)/16
  .00ī0 1ī00   add terms in cycle 5
  .0000 1ī11   p(3) = .01ī        adjust residual, W(5)/32
  .0000 1ī11   p(4) = .01ī0       no adjustment, W(6)/64
  .0000 01ī1   p(5) = .01ī0 1     adjust residual, W(7)/128
  .0000 00ī1   p(6) = .01ī0 1ī    final residual, W(8)/256
```

In cycle 8 above, the second LSD of 0 is attached to the product in accordance with the rounding rules.

3.3. Division and Square-Rooting

In on-line BSD division, yielding $z = x / y$, the residual corresponds to $x - y \times z$; so, not surprisingly, the algorithm resembles that of the multiplication $x = y \times z$, with $x < y$, $W^{(0)} = x$, $z_3 = z_2 = z_1 = z_0 = 0$, and the main recurrence:

$$W^{(i)} = [2W^{(i-1)} + x_{-i} - y_{-i} z^{(i-5)}] - 16 z_{-(i-4)} y^{(i)}$$

Assuming $x < y$, to avoid overflow in fractional division, the real-valued residual $W^{(i)}$ is in $(-16|y^{(i)}| + 2, 16|y^{(i)}| - 2)$. Proper choice of $z_{-(i-4)}$ allows us to maintain this condition, given that $V^{(i+1)}$, composed of $2W^{(i-1)}$ and two terms of magnitude 1 or less, is bounded in $(-32|y^{(i)}| + 2, 32|y^{(i)}| - 2)$ and the last term on the right-hand side of the recurrence can bring the residual back to within the original range. After $z_{-(k-4)}$ has been produced, leaving the residual $W^{(k)}$ in $[-16|y| + 2, 16|y| - 2]$, the last four digits of the quotient z are obtained via the same recurrence, with x and y digits set to 0. The residual $W^{(k+4)}$ holds the rounding information. As in multiplication, the rounding action is based on the round digit $z_{-(k+1)}$ and the sign of the final residual $W^{(k+5)}$.

A direct algorithm, similar to that used for multiplication, is applicable in the case of double-LSD BSD numbers. It proceeds as in normal BSD division until $z_{-(k-4)}$ has been produced, yielding the residual $W^{(k)}$. The extra LSDs, to be examined next, have the same weight as the regular LSDs just processed. Thus, the residual is not left-shifted (or the other terms of $V^{(k+1)}$ in brackets are left-shifted), leading to:

$$W^{(k+1)} = [2W^{(k)} + 2a - 2bz^{(k-4)}] - 16 z_{-(k-3)} y^{(i)}$$

Five more iterations, with a and b digits set to 0 in the preceding equation, produce the last digits $z_{-(k-2)}$, $z_{-(k-1)}$, z_{-k}, round digits $z_{-(k+1)}$, $z_{-(k+2)}$, and final residual $W^{(k+5)}$. Beginning with $W^{(k+1)}$, the scaled residuals have a somewhat wider range of about $(-16|y|, 16|y|)$. Producing an additional round digit $z_{-(k+2)}$ ensures that the true final residual will not exceed $ulp/2$. Other details are quite similar to those of multiplication and thus omitted for brevity.

Computing the square root z of the BSD number x is based on $W^{(0)} = z_1 = z_0 = 0$, and the main recurrence:

$$W^{(i)} = [2W^{(i-1)} + x_{-i}] - 4 z_{-(i-2)} (2z^{(i-3)} + z_{-(i-2)} 2^{-(i-2)})$$

Justification for the on-line delay being equal to two cycles is given in [Erce78]. The second LSD, if present, can be accommodated in a manner similar to division.

Note that unlike on-line addition and multiplication, that can be applied to any pair of operands, both division and square-rooting require scaled operands to work properly. As a result, on-line algorithms for the latter two operations are unsuitable for fixed-point computation.

4. Floating-Point Arithmetic

On-line floating-point arithmetic is complicated by two factors: normalization (which is particularly challenging for radix 2) and rounding. The first of these is indeed serious, as it undermines the basic property of small, constant delay in producing result streams. We have no remedy for this problem [Wata81]. However, we show how the preceding arithmetic algorithms can be adapted to rounding of intermediate results. Without such rounding, floating-point computations may fail to produce valid outputs, even if many guard digits are kept [Parh00].

For our analyses, we assume that BSD significands are (pseudo)normalized to have magnitudes in $[2^{-p}, 1)$, p being a small constant. We then aim for producing enough precision in the result to allow normalizing the significand magnitude to within $[2^{-q}, 1)$, $q \le p$, if desired.

4.1. Addition and Subtraction

In ordinary binary floating-point addition, the operand with the smaller exponent is right-shifted for alignment. The part of the operand that is shifted to the right of the least-significant position is summarized in two guard bits and a sticky bit in order to allow proper rounding in case of a normalizing left shift. To apply the same scheme to double-LSD BSD significands, two issues must be resolved: how to right-shift such numbers and how many extra digits will suffice.

Right-shifting can be done either by physically shifting the extra LSD along with the normal LSD or by using the interpretation of Theorem 1. To derive the number of extra digits needed, we first focus on BSD significands with no extra LSD. The following result applies in this case.

Theorem 2: In adding or subtracting two BSD significands, each pseudonormalized to $[2^{-p}, 1)$, production of $p - q + 1$ guard digits, one round digit, and a sticky digit is necessary and sufficient for retaining enough information to produce a properly rounded result in $[2^{-q}, 1)$.

The extra LSD causes the residual range to grow to $[-3, 3]$ immediately after we have processed the second LSD of the unshifted operand. This necessitates the use of one more round digit. Thus, only one additional operation cycle is needed as a result of the extra LSDs.

4.2. Multiplication

Because fixed-point multiplication of double-LSD BSD numbers needs two round digits (per Section 3.2) and the magnitude of our resultant significand is in $[2^{-2p}, 1)$, production of $2p - q + 2$ extra digits is necessary and sufficient for floating-point multiplication with rounding.

Rather than performing the multiplication as usual and then worrying about normalization, it is possible to defer the production of result digits at the outset as long as both operand digits are 0s or can be recoded as 0s (e.g., 1 followed by –1). In this way, at least one operand will have a magnitude in (1/4, 1). With an h-cycle deferment, the exponent must be decremented by $2h$ and a maximum of $p - q + 4$ extra digits will be required. This latter approach would be advantageous for $p > 2$.

4.3. Division and Square-Rooting

Division of pseudonormalized significands can produce a quotient in $(2^{-p}, 2^{p})$. As the division algorithm presented in Section 3.3 assumed $x < y$, and $z < 1$, production of the quotient digits may have to be delayed by up to p cycles at the outset to ensure $x < y$. The exponent must of course be updated to accommodate this delay. Once output digits begin to emerge, at most two extra digits need to be produced to effect proper rounding.

To apply the square-rooting algorithm of [Erce78], which assumes the radicand x to be in [1/4, 1/2), up to $p/2 - 1$ cycles of delays may be required at the outset. Then, only two extra digits need to be produced.

5. Other Radices and Digit Sets

Our discussion in the previous sections assumed the use of radix-2 BSD numbers. Extension of the results to higher radices is straightforward as long as the digit set used includes both 1 and –1 and is maximally redundant. Note that redundant representations with unsigned digits are specifically excluded [Parh90]. The rounding increment or decrement can be incorporated in the extra LSD as before. As long as the extra LSD is always restricted to [–1, 1], the required changes will be minor. For example the new form of Theorem 1 will dictate that a number having –1, 0, or 1 as its second LSD be infinitely extended with $-(r - 1)$, 0, or $r - 1$ digits, respectively.

Use of higher radices reduces the on-line delays of addition/subtraction, multiplication, division, and square-rooting from 2, 3, 4, and 2 cycles to 1, 2, 3, and 1, respectively [Erce88]. Beyond this, however, going to a higher radix neither complicates, nor simplifies, the algorithms and their guard digit requirements. The use of minimal- or low-redundancy digit sets may necessitate separate analysis and algorithm development (for example, radix 4 numbers with digits in [–2, 2]). On the other hand, on-line algorithms with overredundant digits, of the type produced when several intermediate results are taken as one operand without any reduction applied [Erce99], should be able to handle the extra LSDs with a slight increase in the number of guard digits.

6. Conclusion

We have argued that results of on-line computations based on MSD-first redundant arithmetic can be exactly rounded by incorporating the increment or decrement contribution of the rounding decision as an extra LSD. We showed how numbers with the second LSD attached can be directly subjected to on-line processing in both fixed-point and floating-point formats.

Whereas proper rounding is usually not an issue in signal processing applications which have constituted the main driving force behind the development of on-line arithmetic algorithms, there have been suggestions that on-line arithmetic be used in implementing massively parallel supercomputers, particularly those using optoelectronic computing elements or interconnects (see, e.g., [Fey00]). In such an application context, due attention must be paid to proper rounding of computation results.

References

[Erce78] Ercegovac, M.D., "An On-Line Square Rooting Algorithm," *Proc. Symp. Computer Arithmetic*, Oct. 1978, pp. 183-189.

[Erce87] Ercegovac, M.D. and T. Lang, "On-the-Fly Conversion of Redundant into Conventional Representations," *IEEE Trans. Computers*, Vol. 36, pp. 895-897, July 1987.

[Erce88] Ercegovac, M.D. and T. Lang, "On-Line Arithmetic: A Design Methodology and Applications," *VLSI Signal Processing III* (Proc. IEEE Workshop), 1988, pp. 252-263.

[Erce92] Ercegovac, M.D. and T. Lang, "On-the-Fly Rounding," *IEEE Trans. Computers*, Vol. 41, pp. 1497-1503, Dec. 1992.

[Erce99] Ercegovac, M.D. and T. Lang, "On-Line Scheme for Normalizing a 3-D Vector," *Proc. Asilomar Conf. Signals, Systems, and Computers*, 1999, pp. 1460-1464.

[Fey00] Fey, D. and M. Degenkolb, "Digit Pipelined Arithmetic for 3D Massively Parallel Optoelectronic Circuits," *J. Supercomputing*, Vol. 16, pp. 177-196, 2000.

[Irwi87] Irwin, M.J. and R.M. Owens, "Digit-Pipelined Arithmetic as Illustrated by the Paste-Up System: A Tutorial," *IEEE Computer*, Vol. 20, pp. 61-73, Apr. 1987.

[Oklo82] Oklobdzija, V.G. and M.D. Ercegovac, "An On-Line Square Root Algorithm," *IEEE Trans. Computers*, Vol. 31, pp. 70-75, Jan. 1982.

[Parh90] Parhami, B., "Generalized Signed-Digit Number Systems: A Unifying Framework for Redundant Number Representations," *IEEE Trans. Computers*, Vol. 39, pp. 89-98, Jan. 1990.

[Parh98] Parhami, B. and S. Johansson, "A Number Representation Scheme with Carry-Free Rounding for Floating-Point Signal Processing Applications," *Proc. Int'l Conf. Signal & Image Processing*, Oct. 1998, pp. 90-92.

[Parh00] Parhami, B., *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford, New York, 2000.

[Wata81] Watanuki, O. and M.D. Ercegovac, "Floating-Point on-Line Arithmetic: Algorithms," *Proc. Symp. Computer Arithmetic*, May 1981, pp. 81-86.

[Wata81a] Watanuki, O. and M.D. Ercegovac, "Floating-Point on-Line Arithmetic: Error Analysis," *Proc. Symp. Computer Arithmetic*, May 1981, pp. 87-91.