Approach to the Design of Parity-Checked Arithmetic Circuits

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

Abstract

Achieving fault tolerance via parity checking is attractive due to low overhead in storage and interconnect. However, nonpreservation of parity during arithmetic operations makes it necessary to strip the parity bit before, and to restore it after, such operations. This either leaves the arithmetic part unprotected or else requires complex code conversions. We show that some redundant representations, which are often used for high performance anyway, support a way of designing low-overhead, fault-tolerant arithmetic hardware circuits. An added benefit is localized fault effects due to carry-free arithmetic. Our proposed fault tolerance strategy consists of a way of converting parity-encoded input values to even-parity redundant representations, performing arithmetic with redundant operands in such a way that parity is preserved, and, finally, converting any redundant result to standard parity-encoded output.

1. Introduction

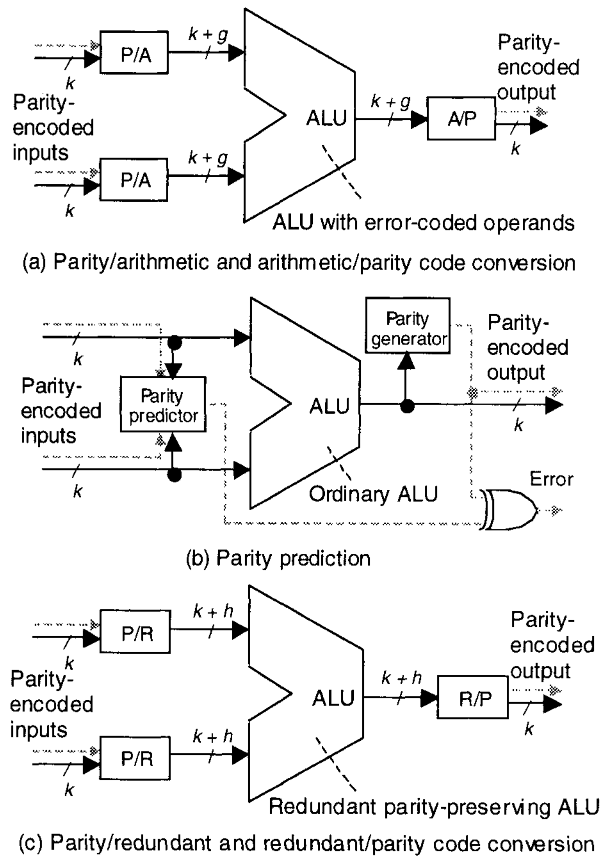
Parity checking was the first error detection method used in digital computers [Bloc48], [Garn58]. Recently, the use of parity checks in conjunction with other hardware schemes such as checkpointing, and software techniques like retry, has gained new momentum [Paul02]. Unfortunately, however, nonpreservation of parity during arithmetic operations makes it necessary to strip the parity bit before, and to restore it after, such operations. This either leaves the arithmetic part unprotected or else necessitates using complex, self-checking code converters [Rao74], [Fuji85], [Parh00], [Lala01] (see Fig. 1a). An alternative is to use a parity prediction circuit (Fig. 1b) which may be quite complex, except in the case of certain simple arithmetic operations such as addition [Hsia63], Fuji81], [Nico93].

Redundant intermediate representations help render parity-checked arithmetic feasible and cost-effective [Parh02]. We thus consider a three-step methodology for checking of arithmetic operations on parity-encoded data (Fig. 1c):

1. Converting even-parity 2's-complement input numbers to intermediate even-parity redundant representations.
2. Performing arithmetic operations on redundant operands in such a way that the even parities are preserved.
3. Converting the even-parity redundant final results to even-parity standard 2's-complement output numbers.

To the extent possible, each step is carried out via local transformations [Thor97]. This is important not only for fault isolation, and hence greater fault tolerance, but also for high performance through parallel processing.

In this paper, we review theoretical results that enable us to perform the aforementioned steps, illustrate how each step might be implemented in hardware, and demonstrate some applications in the design parity-checked adder/subtractors and multipliers. We then briefly discuss how our method can be extended to checking of more complex, arithmetic operations, how checksums can be used in lieu of single parity bits, and how parity checking can be combined with other redundancy methods in hybrid arrangements for greater fault tolerance.



(a) Parity/arithmetic and arithmetic/parity code conversion

(b) Parity prediction

(c) Parity/redundant and redundant/parity code conversion

Fig. 1. Arithmetic with parity-checked data.

2. Even-Parity BSD Encoding

Low-overhead parity checking for arithmetic circuits is made possible by the observation that certain redundant representations possess enough redundancy to allow the generation of the output results with specified parities. Thornton [Thor97] exploited this property to derive a BSD adder design that always produces an even-parity output. This is made possible by a special encoding of the radix-2 BSD digit set {$^{-}1$, 0, 1}, where 0 is assigned two codes with odd and even parities (Table 1). This encoding allows contiguous BSD digits, comprising radix-4 digits in [–3, 3], to be represented in 4 bits with even parity. In other words, the BSD adder can be designed to produce pairs of output digits by means of the encoding of Table 2.

Table 1. Two-bit encoding of BSD digits.

BSD digit: x_i	2-bit code: $y_i z_i$
$^{-}1$	1 1
0	1 0 or 0 0
1	0 1

Table 2. Even-parity code for radix-4 digits.

Radix-4 digit	BSD equivalent	4-bit code
$^{-}3$	$^{-}1$ $^{-}1$	1111
$^{-}2$	$^{-}1$ 0	1100
$^{-}1$	$^{-}1$ 1 or 0 $^{-}1$	0011
0	0 0	1010 or 0000
1	1 $^{-}1$ or 0 1	1001
2	1 0	0110
3	1 1	0101

The encodings shown in Tables 1 and 2 are actually bitwise complements of the ones used by Thornton. We prefer these variants because they correspond to ordinary signed-magnitude interpretation: y is the sign and z is the magnitude, with the two codes for 0 corresponding to ±0. We had originally hoped to be able to assign a weighted bit-set interpretation [Jabe02] to our code, thus allowing the use of standard operations and components in arithmetic circuits; however, this goal does not seem possible, given that the two encodings for 0 differ in one bit, which must thus be zero-weight. Note that the complemented encoding of Table 1 still allows us to use Thornton's adder design with trivial modifications. Other redundant number representations [Aviz61], [Parh90] could be used with the same even-parity encoding provision, but we limit our attention to the BSD representation here.

Based on Table 1, we have devised [Parh02] a three-step process for parity-checked numerical computation: input conversion, arithmetic operations, and output reconversion. A schematic of the proposed approach appears in Fig. 1c. We begin by converting 2's-complement inputs to even-parity BSD format, perform required arithmetic operations on the redundant operands, obtaining all partial results in each stage of the computation with known or easily predictable parities, and, finally, reconvert the output results from BSD to even-parity 2's-complement format. Our method does not imply any additional representational redundancy beyond that needed for BSD operands.

Comparing Figs. 1c and 1a, which are superficially similar, we note that self-checking blocks for parity to arithmetic-code conversion in Fig. 1a are hard to design, slow, and quite complex; by contrast, parity to redundant conversion in Fig. 1c involves only local transformations performed with a handful of gates per digit position, at 2-3 gate delays. Relative ALU costs depend on the arithmetic code used, operations performed, and extent of fault tolerance built in. In general, the ALU of Fig. 1a is likely to be significantly slower, both due to the overhead of error coding and as a result of nonredundant arithmetic. At the final stage, the converters have comparable complexities and delays. Note that when redundant arithmetic is used in the ALU for performance reasons, independent of fault tolerance issues, the added complexity and delay due to format conversion at the last stage of Fig. 1c is insignificant.

3. Input Conversion

We begin by showing how a 2's-complement input number $x = (x_{k-1}x_{k-2}...x_1x_0)_{2\text{'s-compl}}$, of even width k, can be converted to BSD format with the same parity. That is, if an even/odd number of the input bits x_i are 1s, then an even/odd number of the y_i and z_i bits, resulting from encoding the equivalent BSD number according to Table 1, will be 1s. Dividing the number into $k/2$ radix-4 digits, we can encode the resulting radix-4 digits separately, as shown in Fig. 2. For our input conversion process, we use Table 3 to encode the 2 MSBs of the 2's-complement number, one of which is negatively weighted, and Table 4 for all other positions.

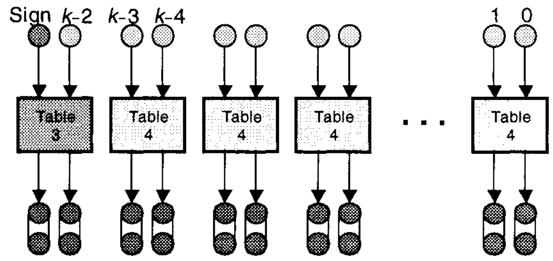


Fig. 2. Converting a 2's-complement input into BSD number with same parity, per the encoding defined in Table 1.

Table 3. Same-parity encoding of 2 MSBs.

x_{k-1} x_{k-2}	y_{k-1} z_{k-1} y_{k-2} z_{k-2}
0 0	d_1 0 d_1 0
0 1	0 d_2 d_2 1
1 0	1 1 1 0
1 1	0 0 1 1

Table 4. Same-parity encoding of bit pairs.

$x_{i+1}\ x_i$	$y_{i+1}\ z_{i+1}\ y_i\ z_i$
0 0	d_3 0 d_3 0
0 1	0 d_4 d_4 1
1 0	0 1 0 0
1 1	0 1 0 1

Table 5. Same-parity encoding of 2 MSBs along with the parity bit *p*.

$p\ x_{k-1}\ x_{k-2}$	$y_{k-1}\ z_{k-1}\ y_{k-2}\ z_{k-2}$
0 0 0	d_5 0 d_5 0
0 0 1	0 d_6 d_6 1
0 1 0	1 1 1 0
0 1 1	0 0 1 1
1 0 0	d_7 0 d_7' 0
1 0 1	1 0 0 1
1 1 0	1 1 0 0
1 1 1	1 d_8 d_8' 1

The entries d_i in Tables 3 and 4 are coupled don't-cares; i.e., both occurrences of each d_i must be assigned the same binary value. For Table 3, choosing $d_1 = d_2 = 1$ yields the simplest circuit realization because it results in $y_{k-1} = x_{k-2}'$ and $y_{k-2} = 1$. However, this would not be a good choice in the context of fault tolerance because y_{k-2}-stuck-at-1 fault will go undetected, raising the risk of fault accumulation. Choosing $d_1 = d_2 = 0$ is somewhat better but may lead to missing a short-circuit between y_{k-1} and z_{k-1}. Both of the remaining choices are acceptable, but $d_1 = 1$ and $d_2 = 0$ leads to a simpler circuit. Similar arguments rule out $d_3 = 0$ as well as $d_3 = 1$, $d_4 = 0$ in Table 4, leaving $d_3 = d_4 = 1$ as the only viable option. Note that in Tables 3 and 4, parity of the bit pair in column 1 is the same as that for the 4 bits in column 2 of each row, leading to parity preservation.

If the 2's-complement input x comes with the parity bit p, we can accommodate p in the encoding of the two MSBs (Fig. 3). Table 5 shows the encoding of x_{k-1} and x_{k-2} so that the parity of the 4 bits $y_{k-1}\ z_{k-1}\ y_{k-2}\ z_{k-2}$ is the same as that of the 3 bits $p\ x_{k-1}\ x_{k-2}$. In this case, there are four coupled don't-care entries and 16 possibilities; $d_5 = d_6 = d_7 = d_8 = 1$ leads to the simplest circuit implementation which is also acceptable in light of fault tolerance considerations.

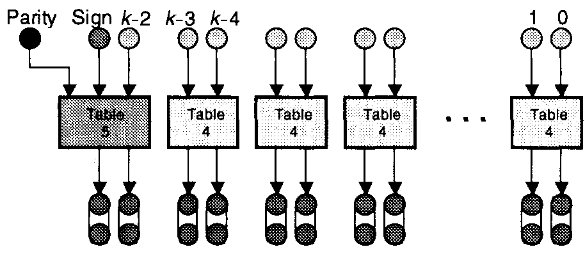


Fig. 3. Converting a 2's-complement input, with parity bit attached, into a BSD number having the same parity.

4. Output Conversion

Consider a k-digit BSD number (k even), with its digits in positions $2i$ and $2i + 1$ encoded per Table 2; i.e., with even parity for each 4-bit group in the $2k$-bit encoding. A simple way of converting a BSD number to 2's-complement is through separating the positive and negative components of the operand and performing subtraction.

So, conversion from BSD to 2's complement can be performed via parity-checked addition [Fuji81], [Nico93], provided that both the positive and negative components are generated with associated even-parity bits. These parity bits, $p_e(x^+)$ and $p_e(x^-)$, are derivable from the $2k$-bit BSD representation of x by noting that x^+ has a 1 in positions where $y_i'z_i = 1$, while $(x^-)'$ has a 1 wherever $y_i' \vee z_i' = 1$. Because k is assumed to be even, the parity of the number of 0s in x^+ and x^- can be determined instead. This strategy makes the inputs supplied to the parity generation trees complements of those used in forming x^+ and x^-, thus improving fault tolerance by avoiding single-point failures and making it less likely that unidirectional multiple errors produce compensating bit inversions.

Parity prediction for an adder's sum output is based on the fact that each carry $c_i = 1$ inverts the parity of the sum bit s_i which in the absence of a carry, would have the same parity as $(x^+)_i \oplus (x^-)_i'$. If p_e is the even-parity function, we have:

$$p_e(s) = p_e(x^+) \oplus p_e((x^-)') \oplus c_0 \oplus c_1 \oplus \ldots \oplus c_{k-1}$$

Note that we cannot use the carries formed in the adder itself to predict the parity of s; doing so would lead to any fault-induced error in c_i inverting both the actual and predicted parities, rendering the error undetectable. This, however, does not imply that the entire carry network, which is responsible for much of the complexity of a fast adder, must be duplicated. Design options with much lower costs are available [Hsia63], [Nico93].

Figure 4 depicts the structure of the converter. Note that the bulk of extra complexity over a standard converter from redundant to 2's-complement lies in the output parity predictor. All other blocks are simple sets or trees of gates.

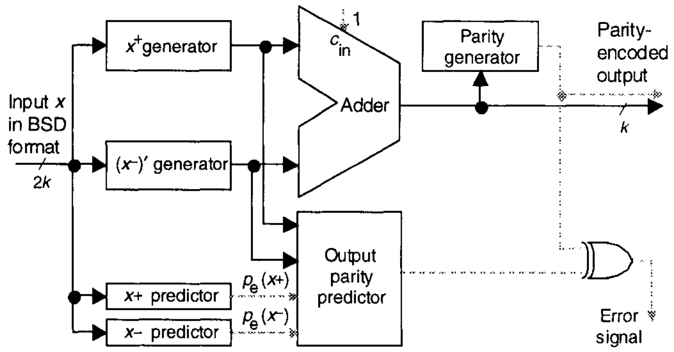


Fig. 4. Parity preserving converter from BSD to 2's complement.

5. Two-Operand Addition

Even though Thornton's adder design, discussed earlier, was offered as an even-parity output adder, it is clear that the adder is in fact a parity-preserving arithmetic block if supplied with even-parity inputs. Both Thornton's original encoding and the complemented one used here also allow for checking of subtraction, given that negation of a BSD number through inverting bit y in the encoding of Table 1 preserves an operand's parity.

For completeness, we present an overview of the parity-preserving BSD adder. In general, restricting transfers to adjacent digit positions leads to a three-stage hardware circuit for BSD addition [Parh00], the result being that the sum digit s_i is formed as a function of three pairs of input operand digits u_{i-2}, v_{i-2}, u_{i-1}, v_{i-1}, u_i, v_i (Fig. 5). The number of signal lines between blocks and the internal block designs vary according to the addition algorithm chosen, leading to tradeoffs between interconnection and circuit complexities. To preserve parity, pairs of adjacent blocks in stage 3 must be merged in view of the coupled generation of output digit pairs with desired parity. Thus, excessive complexity and delay are avoided if we minimize the number of signal lines between stages 2 and 3. The particular algorithm chosen by Thornton satisfies this requirement.

Conceptually, the addition algorithm outlined above might be described in terms of two stages of radix-4 operations proceeding from a position sum in [–6, 6], through interim sum in [–2, 2] and transfer digit value in [–1, 1], to the final sum digit in [–3, 3]. The radix-2 algorithm is simply one particular implementation of the radix-4 version [Jabe02]. This implementation is rather efficient because it relies on weighted posibits, having values in {0, 1}, and negabits, with values in {0, –1}, as well as a small number of variables at each stage.

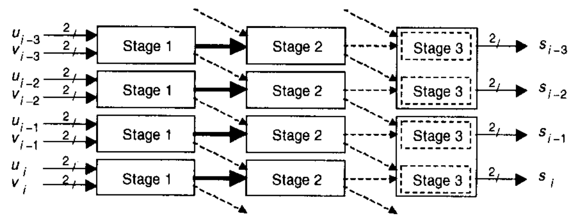


Fig. 5. Parity-preserving BSD adder.

6. Multioperand Addition

Multioperand addition can be performed by repeated use of the two-operand adder discussed in Section 5, provided that the parities of output digit pairs are checked after each addition step, or a small number of steps, to ensure that fault effects remain localized. The parallel (combinational) version of the scheme above consists of a binary tree of two-operand adders, reducing the number of operands by a factor of 2 at each level (Fig. 6). It is also possible to design parallel compressors [Parh00], similar to those for standard binary numbers, to reduce the number of operands to two before adding them in a parity-preserving adder. It is easily seen that three BSD digits cannot be compressed to two BSD digits in a parity-preserving manner. So, compression-based approaches must involve more than three inputs.

In binary multioperand addition, a 2-bit slice of five numbers can be reduced to a 4-bit number via a circuit known as (5, 5; 4)-counter. Figure 7a shows the function of such a unit in dot notation [Parh00]. Figure 7b depicts the BSD counterpart to Fig. 7a. Here, each black-and-white "dot" represents a BSD digit with range of values [–1, 1]. The sum of the five 2-digit BSD numbers, each of which has even parity by assumption, can be represented by two pairs of BSD digits, with each pair having even parity. Thus, the even parity of input data is preserved. One way to realize the (5, 5; 4)-counter of Fig. 7b is to use two copies of the counter in Fig. 7a, one for the positive and another for the negative digit components. The outputs of these two counters are then supplied to a special encoding circuit that ensures even parities for output digit pairs.

It is natural to ask whether larger counters might be applicable here. Unfortunately, the next larger counters of the type shown in Fig. 7b, that is, with parity preservation feature, are impractically large and complex: (85, 85; 8) and (17, 17, 17, 17; 8). These are also hard to design in a way that single faults always lead to isolated output errors. Thus, we advocate the use of (5, 5; 4) parity-preserving counters in multiple levels for reducing larger dot matrices; 2two levels for 11, three levels for 26, and four levels for 65 inputs. It is also possible to reduce 7 BSD digits in the same column to a 3-digit BSD number, as shown in Fig. 5c, while preserving the input parity. However, this method is inferior to that based on Fig. 7b [Parh02].

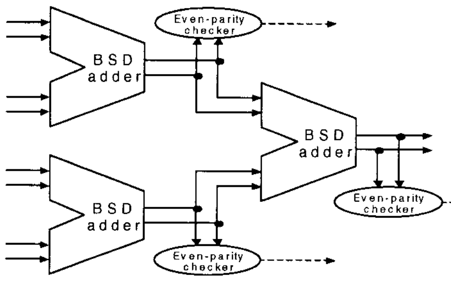


Fig. 6. Binary-tree multioperand addition.

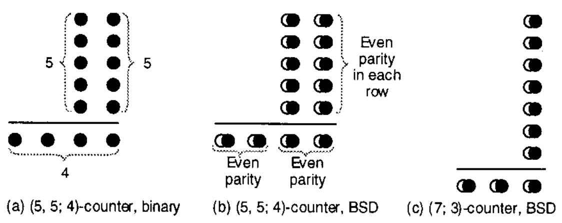


Fig. 7. Parity-preserving compression of BSD operands into 2 or 3 operands.

7. Multiplication

A BSD multiplier can be designed based on sequential shift-add algorithm or parallel tree reduction. In the case of the shift-add scheme, parity preservation is simpler if we perform the multiplication in radix 4, using the following standard recurrence for partial products:

$$s^{(j+1)} = (s^{(j)} + y_j x 4^{k/2}) 4^{-1} \text{ with } s^{(0)} = 0 \text{ and } s^{(k/2)} = \text{product}$$

Radix-4 shifting of BSD numbers encoded as in Table 2 preserves the even parity of digit pairs. The only additional complication over a standard radix-4 multiplier architecture is in the need for a parity-preserving doubling circuit to accommodate $2x$ and $3x$ multiples; as usual, $3x$ is formed as $2x + x$ at the outset or else avoided via digit recoding. The doubling circuit is a highly simplified carry-free BSD adder. Referring to Fig. 5, we note that possible position sum values are in $\{-2, 0, 2\}$, thus reducing the adder to its stage 3, with both bit-inputs at digit position i coming from position $i - 1$. Note that doubling is a form of recoding to preserve the even parity of 4-bit groups after a one-position left shift. With radix-2 multiplication, a doubling or halving scheme must be used in lieu of simple left- or right-shifting of the cumulative partial product.

An alternative to standard radix-4 multiplier architecture outlined above is using the compression scheme of Fig. 7b to combine the cumulative partial product and four other values, two of which are from the set $\{0, x\}$ and the other two from $\{0, -x\}$. This will accommodate the five multiples 0, $\pm x$, and $\pm 2x$, thus implying the need for a recoding circuit to avoid $\pm 3x$. For tree or partial-tree multiplication, a binary tree of limited-carry adders [Parh90] or circuits based on the compression scheme depicted in Fig. 7b can be used. Array multipliers are not as attractive for BSD operands as they are in binary arithmetic, given that the advantages of structural regularity and ease of pipelining are already provided by the much faster tree-based reduction scheme with limited-carry BSD adders.

8. Other Arithmetic Operations

More complex operations can be synthesized from adder and multiplier blocks or else handled directly by producing a number of component terms and combining them using multioperand addition. However, in our case, the relative benefit of merged implementation for complex operations is much less compared to its use with binary arithmetic, given the use of redundant, limited-carry arithmetic.

Arithmetic in a wide array of signal processing applications is dominated by addition and multiplication. Hence, techniques discussed in the preceding sections are adequate for designing parity-checked circuits for many applications. Radix-4 division, square-rooting, and CORDIC algorithms can be added to our list of operations with moderate effort. Modifications to conventional radix-2 or radix-4 division, square-rooting, and other function-evaluation architectures parallel those of multiplication discussed in Section 7.

9. Conclusion

Through redundant BSD representation with inherently even parity, arithmetic operations can be checked against fault induced errors with low circuit redundancy and virtually no added latency, except in the final conversion to nonredundant format. In case redundant representation is used for performance reasons anyway, even the latter overhead becomes insignificant. A similar parity scheme is applicable to carry-save numbers, thus providing an alternative to borrow-save (BSD) representation used in this paper. Whether this alternative leads to any speed-up or simplification remains to be established. The following extensions to this work are currently under investigation:
(1) Possible advatanges of carry-save over BSD encoding;
(2) Use of checksums, also known as generalized parity;
(3) Combination of parity, residue, and other check types;
(4) Parity preservation with nonredundant representation.

References

[Aviz61] Avizienis, A., "Signed-Digit Number Representations for Fast Parallel Arithmetic," *IRE Trans. Electronic Computers*, Vol. 10, pp. 389-400, Sep. 1961.

[Bloc48] Block, R.M. et al, "The Logical Design of the Raytheon Computer," *Math. Tables and Aids to Comp.*, Vol. 3, pp. 286-296, 317-323, Oct. 1948. Also, US Patent No. 2,634,052.

[Ferg99] Ferguson, M.I. and M.D. Ercegovac, "A Multiplier with Redundant Operands," *Proc. 33rd Asilomar Conf. Signals Systems and Computers*, Oct. 1999, pp. 1322-1326.

[Fuji81] Fujiwara, E. and K. Haruta, "Fault-tolerant arithmetic logic unit using parity based codes," *Trans. IECE*, Vol. E64, 1981.

[Fuji85] Fujiwara, H., *Logic Testing and Design for Testability*, MIT Press, 1985.

[Garn58] Garner, H.L., "Generalized Parity Checking," *IEEE Trans. Electronic Computers*, Vol. 7, pp. 207-213, Sep. 1958.

[Hsia63] Hsiao, M.Y. and F.F. Sellers, Jr., "The Carry-Dependent Sum Adder," *IEEE Trans. Electronic Computers*, Vol. 12, pp. 265-268, June 1963.

[Jabe02] Jaberipur, G., B. Parhami, and M. Ghodsi, "Weighted Bit-Set Encodings for Redundant Digit Sets: Theory and Applications," *Proc. 36th Asilomar Conf. Signals Systems and Computers*, Nov. 2002, to appear.

[Lala01] Lala, P.K., *Self-Checking and Fault-Tolerant Digital Design*, Morgan Kaufmann, 2001.

[Nico93] Nicolaidis, M., "Efficient implementations of self-checking adders and ALUs," *Proc. 23rd Int'l Symp. Fault-Tolerant Computing*, pp. 586-595, IEEE Computer Society, 1993.

[Parh78] Parhami, B. and A. Avizienis, "Detection of storage errors in mass memories using arithmetic error codes," *IEEE Trans. Computers*, Vol. 27, pp. 302-308, 1978.

[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.

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

[Parh02] Parhami, B., "Parity-Preserving Transformations in Computer Arithmetic," in *Advanced Signal Processing Algorithms, Architectures, and Implementations XII*, July 2002, to appear.

[Paul02] Paulson, L.D., "Computer System, Heal Thyself," *IEEE Computer*, pp. 20-23, Aug. 2002.

[Rao74] Rao, T.R.N. *Error Codes for Arithmetic Processors*, Academic Press, 1974.

[Rao89] Rao, T.R.N. and E. Fujiwara, *Error-Control Coding for Computer Systems*, Prentice-Hall, 1989.

[Thor97] Thornton, M.A., "Signed binary addition circuitry with inherent even parity output," *IEEE Trans. Computers*, Vol. 46, pp. 811-816, 1997.