Parity-preserving transformations in computer arithmetic

Behrooz Parhami*
Dept. Electrical & Computer Eng., Univ. of California, Santa Barbara

(Invited Paper)

ABSTRACT

Parity checking comprises a low-redundancy method for the design of reliable digital systems. While quite effective for detecting single-bit transmission or storage errors, parity encoding has not been widely used for checking the correctness of arithmetic results because parity is not preserved during arithmetic operations and parity prediction requires fairly complex circuits in most cases. We propose a general strategy for designing parity-checked arithmetic circuits that takes advantage of redundant intermediate representations. Because redundancy is often used for high performance anyway, the incremental cost of our proposed method is quite small. Unlike conventional binary numbers, redundant representations can be encoded and manipulated in such a way that parity is preserved in each step. Additionally, lack of carry propagation ensures that the effect of a fault is localized rather than catastrophic. After establishing the framework for our parity-preserving transformations in computer arithmetic, we illustrate some applications of the proposed strategy to the design of parity-checked adder/subtractors, multipliers, and other arithmetic structures used in signal processing.

Keywords: arithmetic encoding, error code, fault tolerance, parity checking, redundant format, reliable computation

1. INTRODUCTION

Parity checking was the first error detection method used for digital computers. Unfortunately, conventional parity checking is of little use in synthesizing fault-tolerant arithmetic circuits because parity is not preserved during arithmetic operations. One way to use parity for checking of an arithmetic operation is to convert to a different encoding prior to, and reconvert after, the operation. This implies the use of self-checking code converters to ensure that the pre- and post-conversion steps are protected (Fig. 1a). Another way is to utilize a separate parity prediction circuit (Fig. 1b) which may turn out to be quite complex, except for certain simple arithmetic operations such as addition[4].

In this paper, we show that redundant intermediate representations, which are often used for high performance anyway, can help with making parity-checking feasible and cost-effective. Accordingly, we introduce a three-step methodology for checking of arithmetic operations (Fig. 1c):

1. Converting an even-parity 2's-complement number to an even-parity redundant representation.
2. Performing arithmetic operations on redundant numbers in such a way that parity is preserved.
3. Converting an even-parity redundant final result to an even-parity 2's-complement output word.

To the extent possible, each step is carried out using local transformations[9]. This property is important for fault isolation and high performance (parallel processing).

After establishing the theoretical bases for performing the aforementioned steps, we illustrate how each step can be implemented in hardware. Next, we demonstrate the use of our method for checking of adder/subtractors and multipliers, which are the components most frequently used in signal processing systems. In conclusion, we briefly discuss how our method can be extended to error control for other arithmetic operations and how parity checking can be combined with other redundancy methods in hybrid arrangements for greater fault tolerance.

* parhami@ece.ucsb.edu; phone 1-805-893-3211; fax 1-805-893-3262; http://www.ece.ucsb.edu/Faculty/Parhami/index.htm; Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, USA 93106-9560.

2. A FRAMEWORK FOR PARITY-CHECKED ARITHMETIC

An even-parity code is a separable coding scheme that represents a k-bit data word x by attaching a bit p to it, where p is the modulo-2 sum (XOR) of all the bits of x. Any single-bit inversion in the $(k + 1)$-bit even-parity codeword is detectable by a parity checker which is essentially a tree of XOR gates that produces an output of 1 to indicate an error. If we divide the set of $k + 1$ bits of an even-parity codeword into two subsets, the subsets are expected to have the same parity (both even or both odd). This can be the basis of a self-checking parity check circuit: input each subset to a separate tree of XOR gates, complement the output of one tree, and expect to see 10 or 01 at the output for an error-free input (if for any reason, one of the two trees produces an incorrect output, fail-safe operation in ensured due to the appearance of 00 or 11 at the checker output).

Parity checking is not directly applicable to error control in arithmetic circuits, given that parity is not preserved during binary arithmetic. For this reason, special arithmetic error codes have been developed that allow low-redundancy checking of arithmetic operation[7,8]. Here, "low-redundancy" means much lower overhead compared to full circuit duplication. Even for transmission and storage, the effectiveness of parity checking is limited because a pair of bit inversions, as well as a large fraction of other multiple errors, will go undetected. Multiple errors, especially of the unidirectional kind[5], are quite common in modern VLSI circuits due to their failure modes and extremely high densities. Here, we do not concern ourselves with these weaknesses of parity checking. There do exist situations where parity checking provides adequate protection against errors. Furthermore, parity checking need not be the sole error detection mechanism used; hybrid error checking schemes, with parity as one component, are in fact quite common.

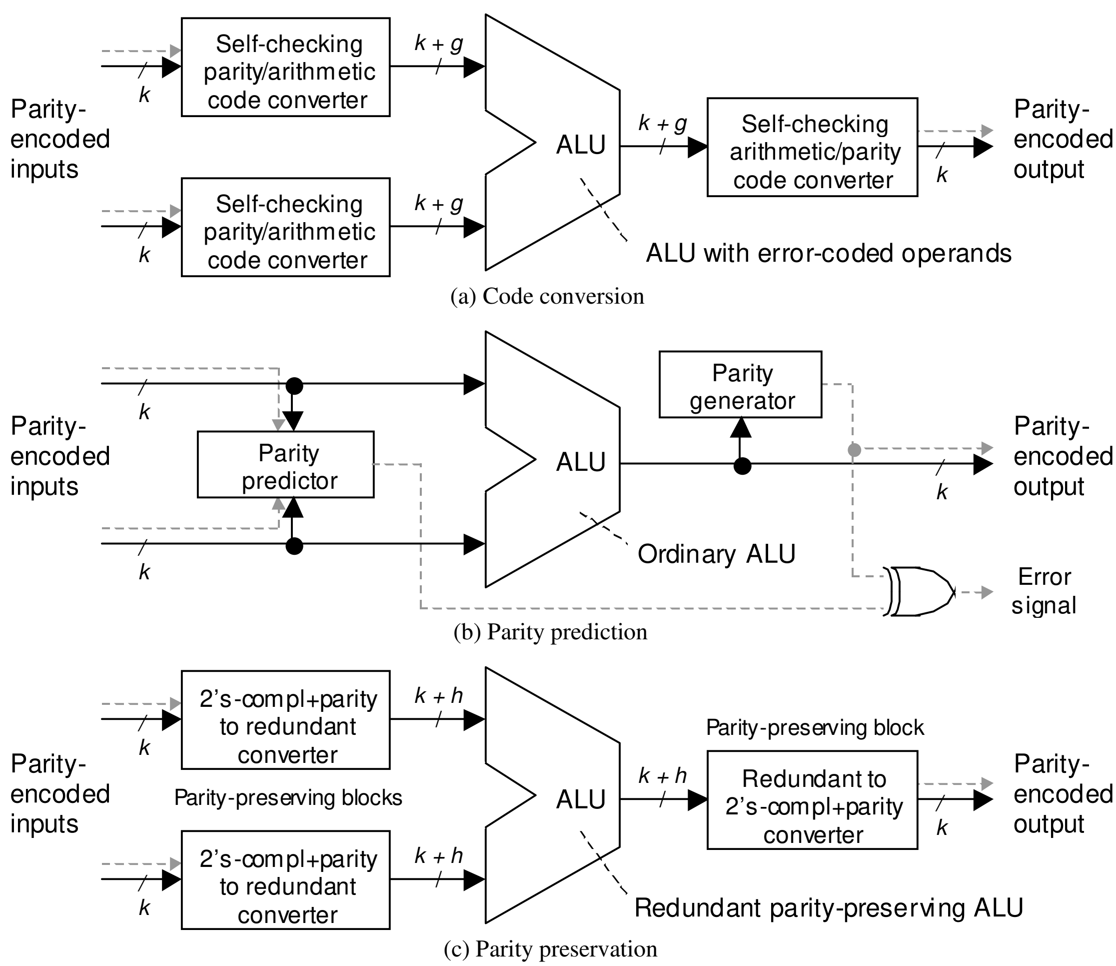


Fig. 1. Three approaches to arithmetic with parity-checked data.

The key insight used to make low-overhead parity checking feasible for arithmetic circuits is that certain redundant representations possess enough redundancy to allow the generation of results with a specified parity. This property was exploited by Thornton[9] in the design of a binary signed-digit (BSD) adder that always produces an even-parity output. This is made possible by encoding the radix-2 digit-set $\{^{-}1, 0, 1\}$ used for BSD representation as in Table 1. We note that the digit value 0 is assigned two encodings, one with odd and the other with even parity, thus allowing pairs of BSD digits, comprising radix-4 digits with values 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 with the 4-bit encoding of Table 2. In this way, each group of 4 bits in the sum, and consequently, the output word as a whole, will have even parity.

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

Note: Negative digit signs are shown as superscripts preceding the digit values to avoid confusion with the binary "minus".

Table 2. Four-bit even-parity encoding of 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 those proposed by Thornton[9]. We prefer these variants because they lead to a conventional signed-magnitude interpretation: y is the sign and z is the magnitude, with the two representations of 0 corresponding to ± 0. Alternatively, y could be viewed as having the weight of -2 and z as having unit weight, which would be consistent with 2's-complement encoding, provided that the digit values were interpreted modulo 2. This latter weighted interpretation of the encoding allows us to subject the digits to arithmetic processing with standard operations and components. Note that the complemented encoding of Table 1 still allows us to use the adder design of Thornton[9] with only trivial modifications. Other redundant representations[6] could be used with the same even-parity encoding provision. However, in this paper, we limit our attention to the BSD redundant representation for the sake of simplicity.

Taking advantage of the encoding of Table 1, we envisage 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. Beginning with even-parity standard 2's-complement inputs, we convert the operands to BSD numbers with even parities (Section 3). We then perform the required arithmetic operations on the redundant operands, obtaining all partial results in each stage of the computation with known or easily predictable parities (Sections 5-6). Finally, we reconvert the output results from BSD to even-parity 2's-complement format (Section 4). Note that the proposed method does not imply any additional representational redundancy beyond that required for BSD-encoded operands of any type. Thus, if BSD or carry-save representation is used for performance reasons anyway, register and wiring costs will not be increased.

On the surface, the scheme of Fig. 1c appears quite similar to that in Fig. 1a. However, there are significant differences between the two. At the first stage, self-checking blocks for parity to arithmetic-code conversion in Fig. 1a are hard to design and quite complex. The resulting circuits also contain many logic levels. By contrast, parity to redundant conversion in Fig. 1c involves only local transformations that can be performed with a handful of gates per digit position and involve a latency of the order of 2-3 gate delays. Within the ALU, relative complexities depend on the arithmetic code used, operations performed, and the extent of fault tolerance built in. Generally speaking, ALU operation in 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. Conversion from redundant representation to 2's-complement involves carry propagation which slows down the process in Fig. 1c. However, if redundant arithmetic is used within the ALU for performance enhancement, independent of fault tolerance issues, then the added complexity and delay due to format conversion at the last stage of Fig. 1c is quite modest.

3. CONVERSION FROM 2'S-COMPLEMENT TO BSD

Consider a 2's-complement number $x = (x_{k-1}x_{k-2} \ldots x_1x_0)_{\text{2's-compl}}$. We show how x can be converted to BSD format with the same parity. That is, if an even/odd number of the input digits x_i are 1s, then an even/odd number of the y_i and z_i digits will be 1s, where $y_i\, z_i$ is the BSD encoding of x_i (see Table 1). Without loss of generality, we assume that k is even. Then, dividing the number into $k/2$ radix-4 digits, we can encode the resulting radix-4 digits separately according to Table 3 (most-significant positions) and Table 4 (all other positions). The reason for treating the most-significant digit differently is that the MSB of a 2's-complement number carries a negative weight. That is:

$$x = (x_{k-1}\, x_{k-2} \ \ldots \ x_1\, x_0)_{\text{2's-compl}} = -x_{k-1}2^{k-1} + (x_{k-2} \ \ldots \ x_1\, x_0)_{\text{two}} = (\bar{x}_{k-1}\, x_{k-2} \ \ldots \ x_1\, x_0)_{\text{BSD}}$$

The entries d_i in Tables 3 and 4 constitute coupled don't-cares; i.e., both occurrences of each d_i must be assigned the same binary value. In 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 $y_{k-2} = 1$ might cause a stuck-at-1 fault on the corresponding circuit line to go undetected, raising the possibility of fault accumulation. Choosing $d_1 = d_2 = 0$ is somewhat better but leads to the danger of missing a short-circuit fault between the two lines carrying y_{k-1} and z_{k-1}. The remaining two choices are both acceptable, but $d_1 = 1$ and $d_2 = 0$ produces a simpler circuit. Applying the same considerations to Table 4, rules out the choice $d_3 = 0$ as well as $d_3 = 1$ and $d_4 = 0$, leaving us with $d_3 = d_4 = 1$ as the only viable option. Note that in each row of Table 3 or Table 4, the parity of the 2 bits in column 1 is the same as that for the 4 bits in column 2, leading to parity preservation at the output. Of course, the magnitude is also preserved.

Table 3. Four-bit same-parity encoding of the two 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. Four-bit same-parity encoding of other 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

Now, if the 2's-complement input x came with a parity bit p, the same bit could serve as the parity bit of the BSD number after encoding according to Tables 3 and 4, making parity prediction trivial. Alternatively, one could take the parity bit p into consideration in the encoding process. The easiest way to do this is to enter p into the encoding of the two most-significant bits, given that the circuit slice for these bits is already different. 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 the parity of the 3 bits $p\ x_{k-1}\ x_{k-2}$. In this case, there are four coupled don't-care entries and 16 possible implementations. The choice $d_5 = d_6 = d_7 = d_8 = 1$ leads to the simplest circuit implementation which is also acceptable in view of fault tolerance considerations discussed earlier.

Table 5. Four-bit same-parity encoding of the two MSBs and 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

As an example, given the 8-bit 2's-complement number 1011 1001, the 8-digit BSD encoding resulting from Tables 3 and 4 is 11 10 01 01 01 00 01 11, where each pair of bits represents one BSD digit according to the encoding of Table 1. The same input number with an even parity bit attached to its left end is 1 1011 1001. Using Tables 5 and 4, we arrive at the BSD encoding 11 00 01 01 01 00 01 11. In the first case, the BSD encoding has the same odd parity as the 8-bit input number; in the second case, the even parity of the input (including its parity bit) has been preserved.

4. CONVERSION FROM BSD TO 2'S-COMPLEMENT

Consider a k-digit BSD number (k even), with its digits in every pair of positions $2i$ and $2i + 1$ encoded according to Table 2; i.e., each group of 4 bits in the $2k$-bit encoding has even parity. The standard way of converting a BSD number to 2's-complement is through subtraction. As shown in the following example, the positive and negative components of the number are separated and used as inputs to a subtractor, which then produces the 2's-complement output.

Input BSD number x :		$^-1$	1	$^-1$	0	1	$^-1$	0	1	
Positive component x^+ :		0	1	0	0	1	0	0	1	$p_e(x^+) = 1$
Negative component x^- :		1	0	1	0	0	1	0	0	$p_e(x^-) = 1$
x^+ in 2's complement :	0	0	1	0	0	1	0	0	1	
x^- inverted to get $(x^-)'$:	1	0	1	0	1	1	0	1	1	
Carry-in c_0 set to 1 :									1	
2's-complement result :	1	1	0	1	0	0	1	0	1	$(x^+) + (x^-)' + 1$

So, conversion from BSD to 2's complement can be performed via parity-checked addition[1,4], provided that each one of the positive and negative components is generated with an even-parity bit attached. These parity bits, $p_e(x^+)$ and $p_e(x^-)$, can be easily derived 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 $y_i' \vee z_i' = 1$. Given that k is even by assumption, the parity of the number of 0s in x^+ and x^- can be determined instead. This will make the inputs to the parity generation trees complements of those used for generating x^+ and x^-, thereby improving fault tolerance by avoiding single-point failures and making it less likely that any unidirectional error will produce compensating bit inversions.

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

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

It is clear 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. However, this observation does not imply that the entire carry network, which is responsible for much of the complexity of a fast adder, must be duplicated. Lower-redundancy options are available[4]. Figure 2 depicts the structure of the converter. Note that the bulk of additional complexity over a standard converter from redundant to 2's-complement is in the output parity predictor block. All other blocks are simple sets or trees of gates.

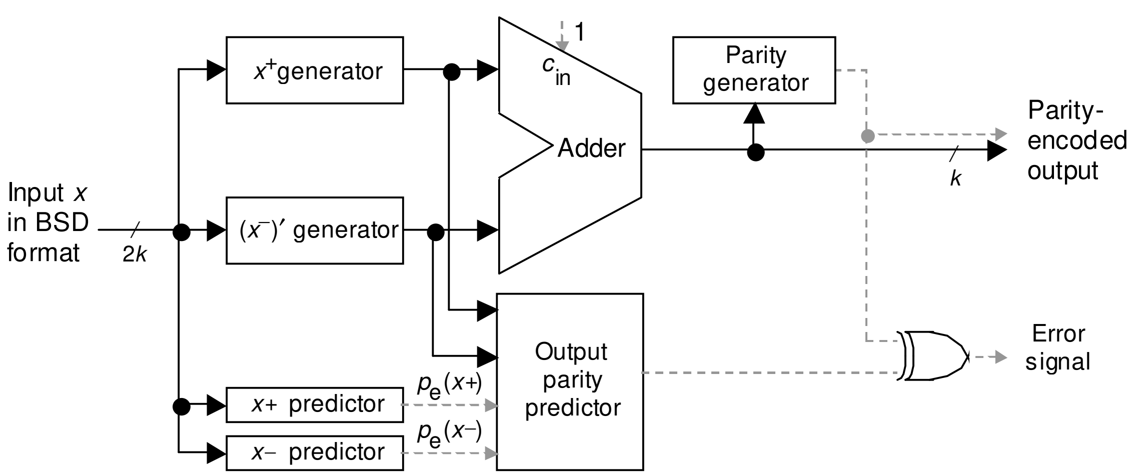


Fig. 2. Schematic diagram of a parity-preserving converter from BSD to 2's complement.

5. PARITY-CHECKED ADDITION / SUBTRACTION

We have already referred to the fact that taking the encoding of Table 1 for BSD digits allows us to use the adder design of Thornton[9] with only trivial modifications. Even though Thornton's design was offered as an even-parity output adder, it is clear that the adder constitutes a parity-preserving arithmetic block if supplied with even-parity inputs. Both the original encoding of Thornton 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 is parity-preserving. Note that parity is preserved during negation with regard to both the 4-bit groups of Table 2 and the complete $2k$-bit encoded word.

For the sake of completeness, we present a brief overview of the parity-preserving BSD adder. In general, a BSD adder requires a three-stage hardware circuit if transfers are to be restricted to adjacent digit positions[7]. This leads to the sum digit s_i being 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. The structure of such a three-stage BSD adder is shown in Fig. 3. The number of signal lines between blocks and the internal design of each block varies according to the addition algorithm chosen. Various choices that are available lead to tradeoffs between interconnection and circuit complexities and allow the designer to derive optimal circuits for particular implementation technologies. Our parity preservation requirement dictates that pairs of adjacent blocks in stage 3 be merged to allow for the coupled generation of output digit pairs. For this reason, it is advantageous to minimize the number of signal lines between stages 2 and 3 in order to avoid excessive complexity and delay caused by a large number of input variables to the solid blocks in stage 3. The particular algorithm chosen by Thornton[9] leads to all lines going from stage 2 to stage 3 carrying one bit (dotted lines in Fig. 3). This algorithm leads to the input digits (4 bits, heaviest lines) being passed between stage 1 and stage 2 horizontally in the same digit slice and a single bit diagonally to the next higher slice.

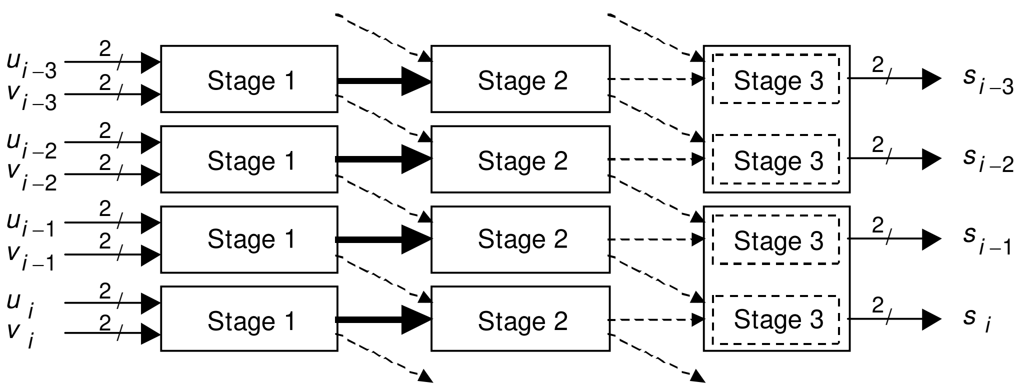


Fig. 3. Schematic diagram of a parity-preserving BSD adder.

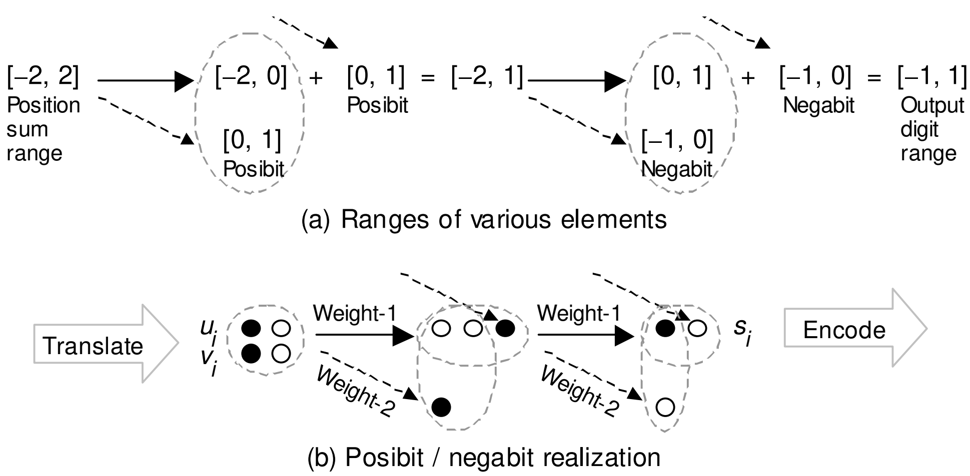


Fig. 4. Limited-carry addition algorithm used in the parity-preserving BSD adder.

The addition algorithm for our parity-preserving BSD adder is shown in Fig. 4a in terms of the ranges of values encountered in each stage. Two BSD input digits together represent a position sum ranging in [–2, 2]. This is decomposed by stage 1 into an interim sum in [–2, 0] and a transfer bit of weight 2 in [0, 1]. Referring to Fig. 3, the position sum need not actually be formed as a separate entity by stage-1 elements. Rather, the input digits can be simply passed on to stage 2 given that they uniquely characterize the actual interim sum in [–2, 0]. The interim sum and transfer entering a box in stage 2 collectively represent a value in [–2, 1]. This latter value is decomposed into a sum posibit and a transfer negabit by stage 2. The inputs to one of the dotted boxes in stage 3 are thus a posibit and a negabit which together represent a digit in [–1, 1]. The role of our stage 3 blocks is thus to translate this (n, p) encoding[7] of a pair of BSD digits into the even-parity encoding of Table 2.

Figure 4b shows one possible implementation of the algorithm above in dot notation using posibits (heavy black dots) and negabits (hollow dots). Each of the two input digits u_i and v_i is encoded by a posibit (u_i^+, v_i^+) and a negabit (u_i^-, v_i^-). The position sum w_i is encoded as two negabits (w_i^-, $w_i^=$). There are two transfers to the next higher position: t_{i+1}^+ is produced in stage 1 and s_{i+1}^- in stage 2. With this notation, and taking into account that stage-3 blocks are coupled as shown in Fig. 3, the complete circuit realization of the adder can be easily derived. Note that with our particular encoding and design, stage 3 of Fig. 3 could be removed were it not for the requirement for parity preservation. This is because the 2 bits representing s_i are combined via concatenation which needs no hardware.

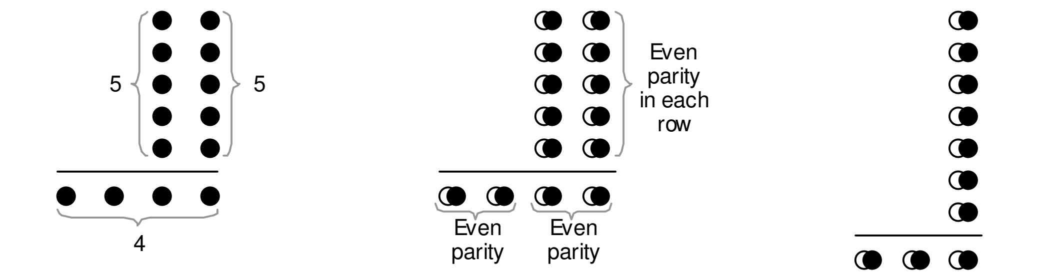
Conceptually, the addition algorithm above could be described in terms of two stages of radix-4 operations proceeding from a position sum in [–6, 6] to interim sum in [–2, 2] and transfer digit value in [–1, 1] to the final sum digit in [–3, 3]. The radix-2 version is simply one particular implementation of the radix-4 algorithm. This implementation is rather efficient because it can be based on weighted bits and a small number of variables at each stage.

6. MULTIPLICATION AND OTHER OPERATIONS

Multioperand addition can be performed by repeated use of the two-operand adder discussed in Section 5; no problem arises as long as the parities of output digit pairs are checked after each addition step to ensure that the effects of faults remain localized. The parallel (combinational) version of this scheme becomes a binary tree of two-operand adders, reducing the number of operands by a factor of 2 at each level. It is also possible to design parallel compressors[7] similar to those used for standard binary numbers to reduce the number of operands to 2 before adding them by means of a parity-preserving adder. Note that 3 BSD digits cannot be compressed to 2 BSD digits in a parity-preserving manner. The reason is that when all 3 digits are 1, encoded as 01 according to Table 1, they collectively have odd parity whereas the 2-digit representation of 3 must have even parity (11 encoded as 0101). Even if this were possible, it would likely be inferior to 2-to-1 reduction offered by limited-carry adders of the type depicted in Fig. 3. So, compression-based approaches must involve more than 3 inputs.

In binary multioperand addition, a 2-bit slice of 5 numbers can be reduced to a 4-bit number by means of a compression circuit known as (5, 5; 4)-counter. Figure 5a shows the function of such a circuit in dot notation[7] as well as its naming rationale. The BSD counterpart to Fig. 5a is shown in Fig. 5b. Here, each black-and-white “dot” represents a 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 by this transformation. One way to implement the (5, 5; 4)-counter of Fig. 5b is to use two copies of the counter in Fig. 5a, one for the positive digit components and another for the negative components. These two counters are then followed by an encoding circuit that ensures even parities for the two output digit pairs.

To compress dot matrices with more than 5 rows of dots, one could opt for larger counters. However, the next larger counters of the type shown in Fig. 5b, that is, with parity preservation feature, are (85, 85; 8) and (17, 17, 17, 17; 8). These are quite complex and also hard to design in such a way that single faults are guaranteed to lead to isolated output errors. Thus, to reduce larger dot matrices, we advocate the use of (5, 5; 4) parity-preserving counters in multiple levels. With 2 levels, 11 inputs can be handled. This number increases to 26 inputs with 3 levels and 65 with 4 levels, which should suffice for practical applications. We note that, due to the impossibility of parity-preserving 3-to-2 reduction of BSD operands, (5, 5; 4)-counters are our fundamental building blocks, in the same way that (3; 2)-counters constitute building blocks for the compression of standard binary numbers.



(a) (5, 5; 4)-counter, binary (b) (5, 5; 4)-counter, BSD (c) (7; 3)-counter, BSD

Fig. 5. Parity-preserving compression of BSD operands into 2 or 3 operands.

It is also possible to reduce 7 BSD digits in the same column to a 3-digit BSD number, as shown in Fig. 5c. In this case, the 3-digit (6-bit) output will be produced so that it possesses the same parity as the 7 incoming digits (14 input bits). That parity can be preserved in this transformation is obvious when the sum of the input digits is in [–6, 6], given that the output will have at least one 0 digit that has both even-parity and odd-parity encodings in Table 1. When the sum is +7, the input parity is odd, which is the same as the parity of 1 1 1 in our BSD encoding (010101). Similarly, when the sum is –7, which implies even parity at the input, the output parity corresponding to $\bar{1}\bar{1}\bar{1}$ in our BSD encoding (111111) matches that of the input. However, this type of compression only guarantees that the final parity will match the collective parity of all input bits. If the number of input bits is very large, then the fault tolerance provided by this scheme is significantly lower than that of Fig. 5b. This drawback, combined with the fact that after a 7-to-3 reduction we are not yet done and need further additions to arrive at a single output word, limits the utility of the (7; 3) compression scheme depicted in Fig. 5c.

As in the case of binary arithmetic, design of BSD multipliers can be based on a shift-add algorithm or tree reduction. For the shift-add scheme, parity preservation is simpler if we perform the multiplication in radix 4; i.e., according to the following recurrence for partial products:

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

Note that radix-4 shifting of BSD numbers encoded as in Table 2 preserves the even parity of BSD digit pairs. The only additional complication over a standard radix-4 multiplier architecture is in the requirement for a parity-preserving doubling circuit to accommodate $2x$ and $3x$ multiples; as usual, the $3x$ multiple is either produced via $2x + x$ at the outset and kept in a register or else avoided through recoding. The doubling circuit is a highly simplified carry-free BSD adder. Referring to Figs. 4a and 3, we note that possible values for the position sum are in {–2, 0, 2}, thus reducing the adder to its stage 3, with both single-bit inputs at digit position i coming from position i – 1. So, the doubling process is essentially a form of recoding to preserve the even parity of 4-bit groups after a one-position left shift. If radix-2 multiplication is implemented, then the aforementioned doubling scheme is used in lieu of simple shifting of the cumulative partial product s.

An alternative to standard radix-4 multiplier architecture (with a doubling circuit) is using the compression scheme of Fig. 5b 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 multiples 0, $\pm x$, and $\pm 2x$, thus implying the need for a recoding circuit to get rid of the $\pm 3x$ multiples. For tree or partial-tree multiplication, either a binary tree of limited-carry adders or circuits based on the compression scheme of Fig. 5b can be used. Array multipliers are not as attractive for BSD operands as they are in binary arithmetic, given that the advantage of structural regularity and ease of pipelining is already provided by the much faster binary-tree-based reduction scheme.

More complex operations can be synthesized from adder/subtractor 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 standard binary arithmetic, given the fact that here arithmetic is redundant and carry-free.

7. CONCLUSIONS

We have shown that through the use of redundant BSD representation with inherently even parity, arithmetic operations can be checked against fault induced errors with fairly low circuit redundancy and virtually no added latency except in the final conversion to nonredundant form. In case redundant representation is used for performance reasons anyway, even the latter overhead becomes insignificant.

Arithmetic in many signal processing applications is dominated by addition and multiplication. Hence, the methods discussed in the preceding sections of this paper are adequate for designing parity-checked circuits for a wide array of practical applications. Radix-4 division, square-rooting, and CORDIC algorithms for transcendental functions can be added to the list with moderate effort. Modifications to conventional radix-2 or radix-4 division and square-rooting architectures parallel those of multiplication discussed in Section 6.

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 leads to any simplification remains to be established. Note that a key to the success of the even-parity encoding scheme used here is our ability to rewrite $\bar{1}\ 1$ as $0\ \bar{1}$ and $1\ \bar{1}$ as $0\ 1$ (Table 2), thus allowing us to take advantage of alternate encodings of 0 to force even parity for the corresponding group of 4 bits. For the digit set [0, 2] of carry-save representation, the rewritings consist of 1 0 in lieu of 0 2 and 1 2 replacing 2 0. Hence, having two encodings for 1, one with even parity and the other with odd parity, would allow us to apply the same strategy with carry-save representation.

Another direction for further investigation is considering parity-preserving transformations with nonredundant representations. For example, one common way of designing fast multipliers is through column compression. It is feasible to compress a column of dots into fewer dots in multiple columns in such a way that the parity is preserved. Given n_0 dots in one column on the input side and $m_{h-1}, \ldots, m_1, m_0$ dots in h columns on the output side, $m_0 \geq 3$ is easily established as a necessary condition for parity preservation. Similarly, we must have $m_1 \geq 2$ or else even powers of 2 cannot be encoded with even parity at the output. All other m_i , $i \geq 2$, can be 1. Whether or not this type of reduction leads to viable multiplier designs remains to be established.

The most common way of error detection in arithmetic operations is through (inverse) residue checking. It is relatively easy to combine parity and residue checking. Note that both parity and residue checks are separable codes. Hence, parity and residue predictions can be done independently and in parallel. Because residue generation is more complicated than parity generation (both are needed at the output side to enable comparison with the predicted value), parity checking will not be on the critical path of the overall computation.

REFERENCES

1. E. Fujiwara and K. Haruta, "Fault-tolerant arithmetic logic unit using parity based codes," *Trans. IECE*, E64, 1981.
2. H. Fujiwara, *Logic Testing and Design for Testability*, MIT Press, 1985.
3. P.K. Lala, *Self-Checking and Fault-Tolerant Digital Design*, Morgan Kaufmann, 2001.
4. M. Nicolaidis, "Efficient implementations of self-checking adders and ALUs," *Proc. 23rd Int'l Symp. Fault-Tolerant Computing*, pp. 586-595, IEEE Computer Society, 1993.
5. B. Parhami and A. Avizienis, "Detection of storage errors in mass memories using arithmetic error codes," *IEEE Trans. Computers*, 27, pp. 302-308, 1978.
6. B. Parhami, "Generalized signed-digit number systems: a unifying framework for redundant number representations," *IEEE Trans. Computers*, 39, pp. 89-98, 1990.
7. B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford Univ. Press, 2000.
8. T.R.N. Rao, *Error Codes for Arithmetic Processors*, Academic Press, 1974.
9. M.A. Thornton, "Signed binary addition circuitry with inherent even parity output," *IEEE Trans. Computers*, 46, pp. 811-816, 1997.