Arithmetic Operators Based on the Binary Stored-Carry-or-Borrow Representation

Daniel Torno [1] and **Behrooz Parhami** [2]
[1] Exorand Technology, Orléans, France ✧ d.torno@computer.org
[2] Univ. of California, Santa Barbara, USA ✧ parhami@ece.ucsb.edu

Abstract

We introduce implementations of arithmetic operators based on the binary stored-carry-or-borrow (BSCB) representation. Several BSCB arithmetic elements, including full-adder, ripple-carry adder, and carry-lookahead adder are presented, followed by detailed design of an array multiplier. In the latter design, the conventional initial AND matrix is transformed and expressed with a redundant radix-2 representation. Each line of the resulting matrix is processed by an accumulation operator with the BSCB representation. Due to a specific property of the multiplication process, this operator is simpler than a standard full-adder cell in terms of gate count, while maintaining the same propagation latency. The entire multiplier is implemented with only XOR and AND gates, thus improving its testability and reliability.

1. Introduction

Redundant number representations allow fast addition by eliminating the carry propagation chains [Aviz61]. Over the years, many researchers have studied and improved upon redundant representations [Parh90], [Phat94], [Taka02]. Redundant representations are used extensively in adder and multiplier implementations, especially those based on the carry-save or borrow-save form. As an example, we cite a multiplication algorithm [Taka85] based on a redundant number expression in radix 2 and the digit set {–1, 0, 1}. Here, we use a redundant radix-2 "binary stored-carry-or-borrow" (BSCB) expression with the digit set {–1, 0, 1, 2}. Ours is similar to the SD3$^{(+)}$ format of Phatak *et al.* [Phat01], but with a different encoding of the digit values.

Elsewhere [Torn09], we introduced half-, full-, ripple-carry, and carry-lookahead adders for the BSCB representation, with a main characteristic of having rather simple Reed-Muller realizations. Some such designs are discussed here, along with a similarly motivated array multiplier. Thanks to their regular structure, array multipliers are VLSI-friendly. The computation delay is proportional to operand widths, but this drawback is counterbalanced by efficient support for pipelining. Multiplier implementations using the carry-save representation have been described in textbooks for at least three decades [Hwan79], [Parh10]. A BSCB array multiplier could be trivially implemented by using BSCB full-adder cells, but the method described here is more efficient. First we show that the matrix of partial products, generated by the AND operation over the multiplier and multiplicand, can be transformed into another matrix of BSCB numbers. Then, we describe the accumulation process, which is similar to the standard one, but with the results expressed in BSCB form. Finally, we deduce Boolean equations for the output signals of the accumulator cell, obtaining a rather direct implementation using XOR and AND gates with the transformation of the initial AND matrix into an XOR matrix. As an example of our scheme, we describe the detailed design of a 5 × 5 array multiplier, along with its performance, complexity, and potential advantages.

2. BSCB Addition

Let capital letters denote integer variables and lowercase letters stand for Boolean variables. Let $S = A + B + C$, with:

$$\begin{cases} A = a_{N-1}\cdot 2^{N-1} + \ldots\ldots + a_n\cdot 2^n + \ldots\ldots + a_1\cdot 2^1 + a_0\cdot 2^0 \\ B = b_{N-1}\cdot 2^{N-1} + \ldots\ldots + b_n\cdot 2^n + \ldots\ldots + b_1\cdot 2^1 + b_0\cdot 2^0 \\ C = c_{N-1}\cdot 2^{N-1} + \ldots\ldots + c_n\cdot 2^n + \ldots\ldots + c_1\cdot 2^1 + c_0\cdot 2^0 \end{cases}$$

A BSCB expression of a three-operand sum can easily be deduced from the carry-save expression. Let the sum S_n be encoded by the Boolean variables r_{n+1} and u_n as follows:

S_n	**–1**	**0**	**1**	**2**
$r_{n+1}\ u_n$	11	00	01	10

The BSCB representation expresses the addition result in the range [–1, 2], a combination of the ranges for carry-save and borrow-save representations.

The Boolean expressions for partial sum s_n and carry cy_n for a standard carry-save full-adder are:

Carry-save addition: $\begin{cases} s_n = a_n \oplus b_n \oplus c_n \\ cy_{n+1} = a_n\cdot b_n \oplus c_n\cdot a_n \oplus c_n\cdot b_n \end{cases}$

By assuming that for each position n, a carry was generated at position $n - 1$, the partial sum u_n is complemented and the "carry" signal r_{n+1} must have a positive action or a negative one in case the assumption was wrong.

BSCB addition: $\begin{cases} u_n = \overline{a_n \oplus b_n \oplus c_n} \\ r_{n+1} = \left(\overline{a_n \oplus b_n}\right)\cdot\left(\overline{b_n \oplus c_n}\right) \end{cases}$

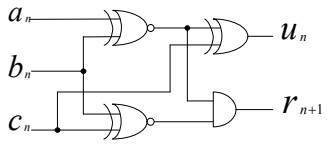

Fig. 1. Design of a BSCB full-adder.

For a 4-bit adder, let cy_{in} and cy_{out} be the input and output carries. We define propagate and generate terms p_n and q_n:

$$p_n = a_n \oplus b_n,\ \ q_n = \overline{b_n \oplus a_{n-1}},\ \ with\ q_0 = \overline{b_0 \oplus cy_{in}},$$

A 4-bit carry-lookahead adder is expressed by:

$$\begin{cases} s_0 = p_0 \oplus cy_{in} \\ s_1 = \overline{a_1} \oplus q_1 \oplus p_0 \cdot q_0 \\ s_2 = \overline{a_2} \oplus q_2 \oplus p_1 \cdot q_1 \oplus p_1 \cdot p_0 \cdot q_0 \\ s_3 = \overline{a_3} \oplus q_3 \oplus p_2 \cdot q_2 \oplus p_2 \cdot p_1 \cdot q_1 \oplus p_2 \cdot p_1 \cdot p_0 \cdot q_0 \\ cy_{out} = a_3 \oplus p_3 \cdot q_3 \oplus p_3 \cdot p_2 \cdot q_2 \oplus p_3 \cdot p_2 \cdot p_1 \cdot q_1 \\ \qquad\qquad \oplus p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot q_0 \end{cases}$$

This structure is very similar to that of a standard carry-lookahead adder and can thus be readily extended to wider operands in the same way with group propagate and generate signals. For a ripple-carry adder cell, we introduce h_{n-1}, a signal propagated from cell $n-1$ to cell n (Fig. 2). Then, the equations for a ripple-carry implementation can be deduced from the previous equations:

$$\begin{cases} s_n = \overline{a_n} \oplus q_n \oplus h_{n-1} \\ h_n = p_n \cdot (q_n \oplus h_{n-1}) \end{cases} with: \begin{cases} q_n = \overline{b_n \oplus a_{n-1}} \\ p_n = a_n \oplus b_n \end{cases}$$

The testability of the proposed BSCB implementations is improved over standard implementations. In the case of a carry-lookahead adder [Kaji97], a set of 8 test vectors is sufficient to detect all single stuck-at faults, thanks to the first stage being composed only of XOR gates.

3. BSCB Mutiplication

Let the multiplier (X) and multiplicand (Y) be N bits wide:

$$X = x_{N-1} \cdot 2^{N-1} + x_{N-2} \cdot 2^{N-2} + ... + x_1 \cdot 2^1 + x_0 \cdot 2^0$$

$$Y = y_{N-1} \cdot 2^{N-1} + y_{N-2} \cdot 2^{N-2} + ... + y_1 \cdot 2^1 + y_0 \cdot 2^0$$

Let the product be a $2N$-bit binary number denoted by P:

$$P = p_{2N-1} \cdot 2^{2N-1} + p_{2N-2} \cdot 2^{2N-2} + ... + p_1 \cdot 2^1 + p_0 \cdot 2^0$$

Assuming the initial AND matrix is computed (see the upper part of Fig. 3), we consider an area of bits set to the value 1 (gray shaded) and generated by the product of bits x_k to x_{k+n-1} with bits y_l to y_{l+m-1}.

The summation of this area A gives following result:

$$A = \left[\sum_{i=k}^{i=k+n-1} x_i \cdot 2^i \right] \cdot \left[\sum_{i=l}^{i=l+m-1} y_i \cdot 2^i \right] = \sum_{i=k}^{i=k+n-1} 2^i \cdot \sum_{i=l}^{i=l+m-1} 2^i$$

$$A = \left(2^{k+n-1} + ... + 2^{k+1} + 2^k\right) \cdot \left(2^{l+m-1} + ... + 2^{l+1} + 2^l\right)$$

$$A = 2^{k+l} \cdot \left(2^{n-1} + ... + 2^1 + 2^0\right) \cdot \left(2^{m-1} + ... + 2^1 + 2^0\right)$$

$$A = 2^{k+l} \cdot \left(2^{n+m} - 2^m - 2^n + 2^0\right)$$

The upper matrix can be transformed into the lower matrix, where the considered area is represented by two rows in the lower matrix, the other rows being zero. The first row is comprised of 3 areas A_1, A_2 and A_3 having respective sums:

$$A_1 = \sum_{i=l+k+n-1}^{i=+\infty} 2^i,\ A_2 = -\sum_{i=l+k-1}^{i=l+k+n-2} 2^i,\ A_3 = \sum_{i=-\infty}^{i=l+k-2} 2^i$$

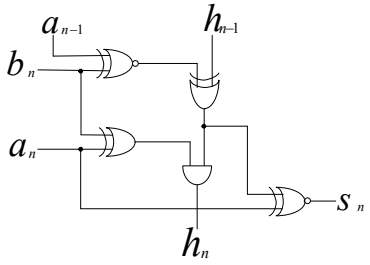

Fig. 2. Design of a BSCB ripple-carry adder.

For the second row, the 3 areas are A_4, A_5 and A_6:

$$A_4 = -\sum_{i=l+m+k+n-1}^{i=+\infty} 2^i,\ A_5 = \sum_{i=l+m+k-1}^{i=l+m+k+n-2} 2^i,\ A_6 = -\sum_{i=-\infty}^{i=l+m+k-2} 2^i$$

The summation of areas A_1 to A_6 gives the same value A, as justified in the following:

$$A_1 + A_4 = \sum_{i=l+k+n-1}^{i=+\infty} 2^i - \sum_{i=l+m+k+n-1}^{i=+\infty} 2^i = 2^{l+m+k+n-1} - 2^{l+k+n-1}$$

$$A_3 + A_6 = \sum_{i=-\infty}^{i=l+k-2} 2^i - \sum_{i=-\infty}^{i=l+m+k-2} 2^i = -2^{l+m+k-1} + 2^{l+k-1}$$

$$A_2 = -\sum_{i=l+k-1}^{i=l+k+n-2} 2^i = -2^{l+k+n-1} + 2^{l+k-1}$$

$$A_5 = \sum_{i=l+m+k-1}^{i=l+m+k+n-2} 2^i = 2^{l+m+k+n-1} - 2^{l+m+k-1}$$

$$\sum_{i=1}^{i=6} A_i = 2^{l+k} \cdot \begin{pmatrix} 2^{m+n-1} - 2^{n-1} - 2^{m-1} + 2^{-1} \\ -2^{n-1} + 2^{-1} + 2^{m+n-1} - 2^{m-1} \end{pmatrix}$$

$$\sum_{i=1}^{i=6} A_i = 2^{l+k} \cdot \left(2^{m+n} - 2^n - 2^m + 2^0\right)$$

Figure 4 shows how an 8 × 8 AND matrix example is transformed into the new matrix E.

	$2^{+\infty}$	...	...	...	...	2^{k+n}	2^{k+n-1}	2^{k+n-2}	...	2^k	2^{k-1}	2^{k-2}	...	...	$2^{-\infty}$
2^{l-1}	0	...	...	...	...	...	0	0	0	...	0	0	0	...	0
2^l	0	...	...	...	...	0	1	1	...	1	0	0	...	...	0
...	0	...	...	...	...	...	...	...	...	...	...	...	...	...	0
2^{l+m-1}	0	...	...	0	1	1	...	1	0	0	...	...	...	...	0
2^{l+m}	0	...	0	0	0	...	0	0	0	...	...	...	...	...	0
2^{l+m+1}	0	...	0	0	0	...	0	0	0	...	...	...	...	...	0

	$2^{+\infty}$	...	...	...	...	2^{k+n}	2^{k+n-1}	2^{k+n-2}	...	2^k	2^{k-1}	2^{k-2}	...	...	$2^{-\infty}$
2^{l-1}	0	...	...	...	...	...	0	0	0	...	0	0	0	...	0
2^l	1	...	...	...	...	1	1	−1	...	−1	−1	1	...	...	1
...	0	...	...	...	...	...	...	...	...	...	...	...	...	...	0
2^{l+m-1}	0	...	...	...	...	...	...	...	...	...	...	...	...	...	0
2^{l+m}	−1	...	−1	−1	1	...	1	1	−1	...	...	...	...	...	−1
2^{l+m+1}	0	...	0	0	...	0	0	...	...	...	...	...	...	...	0

Fig. 3. Transformation of an AND area.

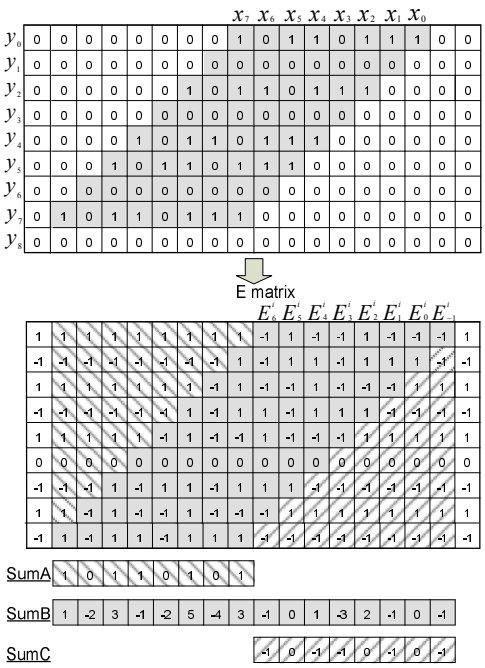

Fig. 4. Transformation of an example AND matrix.

It should be noted that the resulting matrix includes one row and one column more than the initial matrix.

We intend to compute the summation of the transformed matrix (named E) by using the BSCB digit set {–1, 0, 1, 2}. Carry-free addition of two BSCB numbers is known to be impossible. We thus consider the addition of accumulated values expressed in the digit set {–1, 0, 1, 2} with values of the matrix E expressed in the digit set {–1, 0, 1}. In matrix E of Fig. 4, three different areas are added: area A, area B (gray shaded), and area C. The two others at the extreme left and right sides (left blank) have always a zero sum. The three lines SumA, SumB, SumC show the summation result of the three respective areas. The SumA (respectively, SumC) line is composed of values between 0 and 1 (0 and –1); these two cases will be treated further. We now focus on the summation process for SumB. Let E^i be the ith row of the transformed matrix and let L^i be the ith iteration of the accumulated sum. The initial accumulated sum L^0 is initialized with the value E^0 of the first row of the transformed matrix. We introduce a transfer digit T^i from the set {–1, 0, 1} whose value is related to L^i: a carry (respectively, borrow) is propagated to the next stage $n + 1$ if the accumulated value is 2 (respectively, –1). The resulting accumulated sum of iteration $i + 1$ is defined by: $L_n^{i+1} = L_n^i + T_n^i - 2 * T_{n+1}^i + E_{n+1}^{i+1}$

The matrix E is not composed of random values, but of values related to x and y values, thus resulting in a regular structure of values within the digit set {–1, 0, 1}. According to the E values we examine how the accumulated sum is computed.

Let E_{n+1}^{i+1} be the value at row $i + 1$ and column $n + 1$ of the matrix E, and let e_{n+1}^{i+1} and z_{i+1} be two Boolean variables such that: $e_{n+1}^{i+1} = \overline{x_{n-i} \oplus y_{i+1}}$ $z_{i+1} = y_{i+1} \oplus y_i$

E_{n+1}^{i+1} can be expressed in the digit set {–1, 0, 1} according to:

E_{n+1}^{i+1}	**–1**	**0**	**1**
$e_{n+1}^{i+1}\ z_{i+1}$	11	x0	01

To perform the sum computation, we distinguish accumulated values L_n^i and E matrix values E_{n+1}^{i+1} in terms of their respective values e_n^i with initial values $L_n^0 = E_{n+1}^0$.

Table I. Accumulation cases.

			$L_n^i = -1$ *or* $L_n^i = 1$				$L_n^i = 0$ *or* $L_n^i = 2$			
				$L_{n-1}^i\ (e_n^i = 0)$				$L_{n-1}^i\ (e_n^i = 0)$		
L_n^{i+1}				0	1	2		0	1	2
			$L_{n-1}^i\ (e_n^i = 1)$				$L_{n-1}^i\ (e_n^i = 1)$			
E_{n+1}^{i+1}	e_{n+1}^{i+1}	z_{i+1}	–1	0	1		–1	0	1	
–1	1	1		0	0	1		–1	–1	0
0	0	0		1	1	2		0	0	1
1	0	1	1	2	2		0	1	1	
0	1	0	0	1	1		–1	0	0	

The underlying recurrence relation is defined as:

$$\forall i,n \ \ if: \begin{cases} e_n^i = 0 \Rightarrow L_{n-1}^i \in \{0,1,2\} \\ e_n^i = 1 \Rightarrow L_{n-1}^i \in \{-1,0,1\} \end{cases} \qquad then: \begin{cases} e_{n+1}^{i+1} = 0 \Rightarrow L_n^{i+1} \in \{0,1,2\} \\ e_{n+1}^{i+1} = 1 \Rightarrow L_n^{i+1} \in \{-1,0,1\} \end{cases}$$

It comes from the relation:

$$e_{n+1}^{i+1} = e_n^i \oplus z_{i+1} \quad (e_{n+1}^{i+1} = \overline{x_{n-i} \oplus y_{i+1}} = \overline{x_{n-i} \oplus y_i \oplus y_i \oplus y_{i+1}})$$

Cases in striped boxes never occur (they are "don't cares"). The recurrence relation allows the BSCB accumulation of all cases over the matrix by keeping the accumulated sum in the digit set {–1, 0, 1, 2}.

The accumulated value L_{n-1}^i is expressed according to the following encoding, where r_n^i and u_{n-1}^i are Boolean variables:

L_{n-1}^i	**–1**	**0**	**1**	**2**
$r_n^i\ u_{n-1}^i$	11	00	01	10

For the parity u_n^{i+1} (resp., the carry-or-borrow r_{n+1}^{i+1}), Table I is transformed into the Karnaugh map of Table II (resp., III).

Table II. Parity Karnaugh map.

u_n^{i+1}		1				0				u_n^i
e_{n+1}^{i+1}	z_{i+1}	11	00	01	10	11	00	01	10	$r_n^i u_{n-1}^i$
1	1		0	0	1		1	1	0	
0	0		1	1	0		0	0	1	
0	1	1	0	0		0	1	1		
1	0	0	1	1		1	0	0		

Table III. Carry-or-Borrow Karnaugh map.

r_{n+1}^{i+1}		1				0				u_n^i
e_{n+1}^{i+1}	z_{i+1}	11	00	01	10	11	00	01	10	$r_n^i u_{n-1}^i$
1	1		0	0	0		1	1	0	
0	0		0	0	1		0	0	0	
0	1	0	1	1		0	0	0		
1	0	0	0	0		1	0	0		

The accumulation operation is :

$$\begin{cases} u_n^{i+1} = u_n^i \oplus r_n^i \oplus z_{i+1} \\ r_{n+1}^{i+1} = \left(z_{i+1} \oplus r_n^i\right) \cdot \left(e_{n+1}^{i+1} \oplus u_n^i\right) \end{cases}$$

The following "don't care" cases are not possible:

$r_n^i = 1$ and $u_{n-1}^i = 1$ and $e_{n+1}^{i+1} \oplus z_{i+1} = 0$

$r_n^i = 1$ and $u_{n-1}^i = 0$ and $e_{n+1}^{i+1} \oplus z_{i+1} = 1$

This can be synthezised by the equation:

$r_n^i \cdot \left(z_{i+1} \oplus e_{n+1}^{i+1} \oplus u_{n-1}^i\right) = 0$

Or with a substitution: $r_n^i \cdot \left(e_n^i \oplus u_{n-1}^i\right) = 0$

This equation is a corollary of the recurrence relation. At each stage i the accumulated sum L_{n-1} (in terms of the Boolean variables r_n and u_{n-1}) not only expresses the sum but it also conveys information about the multiplier and the multiplicand values through the relations:

$L_{n-1}^i = 2$ ($r_n^i = 1$ and $u_{n-1}^i = 0$) only if: $\overline{x_{n-i} \oplus y_i} = 0$

$L_{n-1}^i = -1$ ($r_n^i = 1$ and $u_{n-1}^i = 1$) only if: $\overline{x_{n-i} \oplus y_i} = 1$

This property can be used to check the correctness of all accumulated sums.

Figure 6 shows a 5 bit by 5 bit implementation of the BSCB multiplication process described above. There are 3 different stages, detailed in Fig. 5:

INIT cells process the initial values,

ACC cells compute the accumulated sums,

RCA cells convert from BSCB to standard binary form.

The left side of the matrix (area SumA in Fig. 4) is added by setting the u_n^i input of the left diagonal of the ACC cells to y_i. The right side of the matrix (area SumC) is added by using a dedicated right diagonal of cells with e_n^i set to y_{i-1}.

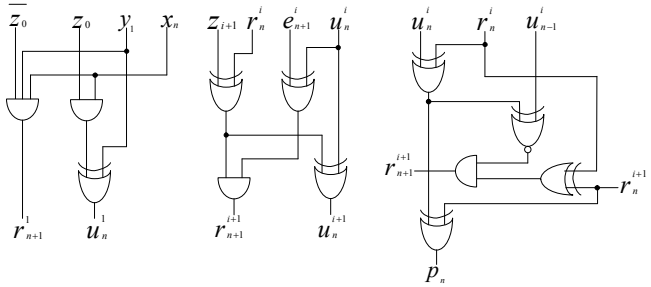

Fig. 5. INIT, ACC, and RCA cells.

Propagation delay for an implementation without pipelining:

The propagation delay of the ACC cell is equivalent to two XOR gates, thus identical to the propagation delay of a standard full-adder (also two XOR gates for the worst case path depending on the technology and the design). An N-bit standard array multiplier is realized with $(N-1)^2$ full-adders and the propagation delay from the partial product to final stage (RCA or CLA) is equivalent to $2(N-1)$ gates.The propagation delay of the BSCB array multiplier is equivalent to $2N$ gates, that is, 2 gates more than the standard array multiplier whatever the operand widths.

Propagation delay for an implementation with pipelining:

The BSCB array multiplier is suitable for pipelined VLSI implementation and should allow an equivalent or better throughput over the standard one, given that the propagation delay of the standard full-adder is two or three gates (worst cases: two XOR gates or one XOR gate and two NAND gates, depending on the technology) instead of two XOR gates for the BSCB ACC cell.

A 5 × 5 array multiplier is shown in Fig. 6. In spite of the overhead due to the extra row and diagonals of ACC cells, beginning with operand width of 16 bits, the BSCB array multiplier needs fewer gates (Table IV), because the ACC cell is realized with 4 gates instead of 5 gates for the standard full-adder (assuming that only 2 inputs gates are used and that all kinds of gates have the same complexity in term of area).

Testability of the cells is improved by the use of XOR and AND gates. For all cells (INIT, ACC, RCA) we obtain a 100% coverage for single stuck-at faults with 4 test vectors.

Table IV. Binary and BSCB array multipliers compared.

Operands bit width	8	16	32	64	128
Standard Array Multiplier	344	1456	5984	24256	97664
BSCB Array Multiplier	380	1404	5372	20988	82940
Gate count difference in %	+10.4	–3.6	–10.2	–13.5	–15.0

4. Conclusion

New implementations of binary adders were proposed based on the binary stored-carry-or-borrow representation. It seems that this BSCB representation leads rather directly to Reed-Muller implementations of all known types of adders (half-, full-, ripple-carry, carry-lookahead adders). A direct advantage of this kind of implementation is that the testability is improved over known implementations due to the use of XOR gates, as demonstrated by prior work. A related drawback is that the rise in switching activity leads to an increase in power consumption.

A number of areas merit further exploration. One is the derivation of new designs of XOR gates. A second area is that of incorporating fault tolerance features in the adder designs. For example, carry-lookahead adders might be checked

through parity prediction. It is evident that the parities of the two inputs can be used to generate the parities of the signals generated by the first XOR stage in the adder. A third area for further investigation is implementation with reversible logic, perhaps using the method of parity preservation [Parh06] for fault tolerance. These adders are particularly suitable for reversible-logic implementation due to the fact that more than 3/4 of their circuits consist of XOR gates.

We also presented an array multiplier based on the BSCB representation. Like ordinary array multipliers, our design is VLSI-friendly and easily pipelined, with the added benefit of simpler cells and lower overall latency. The multiplication process is very similar to the process of the standard array multiplier. Accumulation operations are performed without carry propagation, but all intermediate accumulated sums are expressed in the BSCB redundant representation instead of the carry-save representation. Our architecture shows two main characteristics: (1) The standard initial AND matrix is replaced with an XOR matrix, and (2) A recurrence relation between the XOR products and the partial accumulated sums makes the implementation simpler and more reliable.

References

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

[Hwan79] K. Hwang, *Computer Arithmetic: Principles, Architecture and Design,* Wiley, pp. 69-96, 1979.

[Kaji97] S. Kajihara and T. Sasao, "On the Adders with Minimum Tests," *Proc. 5th Asian Test Symp.,* 1997.

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

[Parh06] B. Parhami, "Fault-Tolerant Reversible Circuits," *Proc. 40th Asilomar Conf. Signals, Systems, and Computers*, 2006, pp. 1726-1729.

[Parh10] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford, 2nd ed., 2010.

[Phat94] D. S. Phatak and I. Koren, "Hybrid Signed-Digit Number Systems: A unified Framework for Redundant Number Representations With Bounded Carry Propagation Chains," *IEEE Trans. Computers*, Vol. 43, pp. 880-891, August 1994.

[Phat01] D. S. Phatak, T. Goff, and I. Koren, "Constant-Time Addition and Simultaneous Format Conversion Based on Redundant Binary Representations," *IEEE Trans. Computers*, Vol. 50, pp. 1267-1278, November 2001.

[Taka85] N. Takagi, H. Yasuura, and S. Yajima, "High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree," *IEEE Trans. Computers*, Vol. 34, pp. 789-796, September 1985.

[Taka02] N. Takagi, "Multiple-Value-Digit Number Representations in Arithmetic Circuit Algorithms", *Proc. 32nd IEEE Int'l Symp. Multiple-Valued Logic*, 2002.

[Torn09] D. Torno, "Reed-Muller Adders Based on Binary Stored Carry or Borrow Representation," *Proc. 18th Int'l Workshop Post-Binary ULSI Systems*, pp. 56-65, 2009.

[Zimm98] R. Zimmermann, *Binary Adders Architecture for Cell-Based VLSI and Their Synthesis,* Hartung-Gore, 1998.

Fig. 6. Design of a 5 × 5 BSCB array multiplier.