Transformations of Signed-Binary Number Representations for Efficient VLSI Arithmetic

Boris D. Andreev, Edward L. Titlebaum, and Eby G. Friedman

Department of Electrical and Computer Engineering
University of Rochester
Rochester, New York 14627
{bandreev, tbaum, friedman} @ece.rochester.edu

Abstract

The VLSI implementation of arithmetic operations may be significantly improved by using non-conventional number representations and transforming intermediate results from one format to another format. For a target function, the objective is to change the number representations of the input and output operands such that a minimum amount of logic circuitry is required to achieve a computation. Redundant arithmetic has received increasing interest in the past decade to reduce or eliminate carry propagation chains. The development of an analytical framework that expands the scope of functions that can be efficiently implemented using signed-binary representation is discussed in this paper. Implementation details are described that demonstrate the application of these results. Particular attention is placed on realizing the (a+b), –(a+b), (a–b), and –(a–b) functions in a complex ±1 multiplier serving as a pseudonoise code scrambler in wireless CDMA transceivers.

1. Introduction

The speed of arithmetic logic circuits is a primary characteristic in many digital VLSI systems, and is often achieved at the expense of increased area or power dissipation. The exploding growth of portable devices with severe constraints on the available resources has motivated a corresponding interest in innovative design approaches that overcome tradeoffs among area, speed, and power.

One of the most common VLSI circuits is the binary adder. Carry signal propagation through long chains of logic, as in the case of a conventional ripple-carry adder, is a major source of performance degradation. In many complex arithmetic circuits, several structures with the complexity of an adder are required, leading to significant delay.

During the past decade, redundant arithmetic has received increasing interest due to the ability to reduce or eliminate carry propagation chains. Parallel addition is performed by selecting an intermediate representation of the sum of two numbers $a+b$ such that the final result is obtained using simple logic without need for carry propagation. Although addition with redundant arithmetic techniques may offer significant improvements in computing speed, efficient circuit implementations have traditionally been difficult to achieve. Since input and output operands of arithmetic circuits are often required in two's-complement format, conversion circuits to/from the intermediate representations are needed. These interface circuits degrade the overall improvement in speed - the conversion delay overhead must be smaller than the delay reduction achieved using parallel computation techniques. For these reasons, many systems for fast arithmetic, such as the residue number system (RNS) [1] and the logarithmic number system (LNS) [2], have not received widespread use because of the significant overhead of the conversion process. Alternatively, the number representations proposed in this paper may be transformed relatively easily to/from two's-complement format using the transformations described in the following sections.

The realization of four functions, *(a+b), –(a+b), (a–b),* and *–(a–b)*, with minimum resources is discussed in this paper. These functions are used in a complex pseudonoise (PN) code scrambler for wireless third generation CDMA transceivers [3, 4]. The scrambler multiplies the complex input signal $a + jb$ by the PN code $PN_{re} + jPN_{im}$ to obtain the spread spectrum output $A + jB = (a + jb) \cdot (PN_{re} + jPN_{im})$. Based on knowledge of the particular pseudonoise code, the signal of the desired user is extracted from the multiuser interference. Transformations of number representations for an efficient VLSI implementation of the operations in the PN scrambler are described in this

[*] This research is supported in part by the Semiconductor Research Corporation under Contract No. 99-TJ-687 and No. 2003-TJ-1068, the DARPA/ITO under AFRL Contract F29601-00-K-0182, grants from the New York State Office of Science, Technology & Academic Research to the Center for Advanced Technology – Electronic Imaging Systems and to the Microelectronics Design Center, and by grants from Xerox Corporation, IBM Corporation, Intel Corporation, Lucent Technologies Corporation, Eastman Kodak Company, and Photon Vision Systems, Inc.

paper. For a particular arithmetic function, the objective is to change the number representations of the input and output numbers such that a minimum amount of logic circuitry is required to achieve the computation.

Of all redundant sets, the *signed-binary* (SB) set $S_x = \{0, 1, \overline{1}\}$ and the *initial sum* set $S_y = \{0, 1, 2\}$ have received significant attention due to the small size, relative ease of representation within the binary number system, and low conversion overhead to/from the conventional two's-complement format. The sum of any two bits a_i+b_i may be represented by a digit y_i in the *initial sum* set $S_y = \{0, 1, 2\}$. The sum of two N-bit numbers $a + b$ may therefore be expressed as an (N+1)-digit *initial sum* number y, with digits $y_i \in S_y$. The digits from the two sets, S_x and S_Y, are related through the self-inverting transformation $tr_{xy}(z_i) = 1 - z_i$, $z_i \in \{S_x \cup S_y\}$ [5]. The numerical values of some N-digit numbers from both sets may be expressed as

$$T_x(x) = \sum_{i=0}^{N-1} x_i 2^i \quad \text{and} \quad T_y(y) = \sum_{i=0}^{N-1} y_i 2^i \,. \tag{1}$$

In the following discussion, the notations $T_x(x)$ and $T_Y(y)$ are used to denote both the numerical values of the corresponding number, x or y, as well as the (N+1)-bit representation of these values in two's complement binary format. Note that the value $T_Y(y)$ is equal to the sum of $a+b$; only the number format of y is not binary. As shown in [5], if an initial sum y is transformed into a SB number x through the digit transformation tr_{xy}: $y_i = 1 - x_i$, the relation between the two's-complement numbers is

$$\begin{aligned} a+b = T_y(y) = T_y[tr_{xy}(x)] &= \sum_{i=0}^{N-1} tr_{xy}(x_i) 2^i \\ &= 2^N - 1 - T_x(x) = 2^N + \overline{T_x(x)} \quad , \end{aligned} \tag{2}$$

where $-T_x(x) = \overline{T_x(x)} + 1$ and $\overline{T_x(x)}$ is the binary number $T_x(x)$ with all bits inverted. This relation may be applied to achieve an alternative VLSI implementation of an adder by mapping the bits of the two N-bit numbers to signed-binary digit from S_x (skipping the initial summation to y_i), converting the redundant number into the two's-complement counterpart $T_x(x)$, and finally, transforming that result into the sum of the numbers $T_y(y)$ [5]. The conversion of an SB number to TC is achieved through the reverse application of (2) (tr_{xy} is symmetric). This approach permits the optimization of more flexible arithmetic circuits when certain manipulations of the intermediate results (or several consecutive operations) are applied to perform carry-free addition in the S_x - S_Y domain. Efficiency is achieved because expensive operations in the two's-complement domain are performed with less resources expended on the intermediate signed-binary number which is easily manipulated (inverted and/or added) without a carry propagation delay. The SB number is mapped onto the correct result in two's-complement. The most resource intensive part of this process is the SB→TC conversion, which is essentially identical to a conventional two's-complement addition [5]. $T_x(x)$ becomes the sum of the numbers $T_Y(y)$ through an inversion of all but the last bit as described by (2).

An analytical description of the remaining arithmetic functions in a PN scrambler is provided in section 2. The sign bit is discussed in section 3 while numerical examples that demonstrate the application of these results are presented in section 4. To illustrate the general framework of this methodology, alternative transformations are considered in section 5. A practical application of these transforms for wireless CDMA transceivers is described in section 6. Concluding remarks are offered in section 7.

2. Analytical Expressions of the Functions of a Complex ±1 Mulitplier

The objective of this section is to apply the ideas briefly presented in the introduction to achieve expressions for the functions *–(a+b), (a–b),* and *–(a–b)*. Based on (2), *–(a+b)* can be expressed as

$$-(a+b) = -T_y(y) = -\left(2^N - 1 - T_x(x)\right) = -2^N + 1 + T_x(x)\,. \tag{3}$$

Conventional addition of 1 requires a carry propagation chain, making the *–(a+b)* function difficult to implement. Any addition of a signed-binary number with a two's-complement number, however, may be completed in two gate delays, producing a signed-binary output [6]. An algorithm to implement the *–(a+b)* function is:

1. Sum the two input operands bitwise, producing an initial sum in the set $S_Y = \{0,1,2\}$.
2. Map this result to a signed-binary number using $x_i = (1-y_i)$, $x_i \in S_x = \{0, 1, \overline{1}\}$.
3. Perform the addition of +1 in two gate delays. The result is in borrow-save signed-binary form [6, 7].
4. Convert the result from SB to TC format using a conventional adder [5, 6, 8].

Note that the first three operations are simple logic functions over a limited number of input operands, making this algorithm amenable to optimization. It is assumed that the result of the +1 addition is required to be in sign-magnitude format. A number in signed-magnitude format can be efficiently transformed and passed to the carry generate-propagate (*G-P*) inputs of a carry lookahead adder (CLA), where the conversion to TC is performed [9]. Applying conventional logic optimization, the (N+1)-digit signed-binary result corresponding to $T_x(x)+1$ is

$$d_0'' \begin{cases} S_0'' = \overline{M_0'} = a_0 \oplus b_0 \\ M_0'' = \overline{M_0'} = a_0 \oplus b_0 \end{cases} \tag{4}$$

$$d_1'' \begin{cases} S_1'' = M_1' \cdot S_0' = \overline{a_1 \oplus b_1} \cdot a_0 b_0 \\ M_1'' = \overline{M_1' \oplus S_0'} = \overline{a_1 \oplus b_1} \oplus a_0 b_0 \end{cases} \tag{5}$$

Table 1: Summary of transforms and implementation of the four required functions

Function	Relation to $T_Y(y)$	Relation to $T_x(x)$	Implementation description
a+b	$T_Y(y)$	$2^N + \overline{T_X(x)}$	1. Obtain all x_i from a_i, b_i 2. Produce the 2's complement $T_x(x)$ 3. Invert all bits, set the Nth bit
a+b	$T_Y(y)$	$2^N - (1 + T_X(x))$	1. Obtain all x_i from a_i, b_i, with +1 and inverted sign bits 2. Produce the 2's complement $-(T_x(x)+1)$, set the Nth bit
–(a+b)	$-T_Y(y)$	$-2^N + 1 + T_X(x)$	1. Obtain all x_i from a_i, b_i with +1 2. Produce the 2's complement $T_x(x)+1$, set the Nth bit
(a–b)	$T_Y(y)+1$	$2^N - T_X(x)$	1. Obtain all x_i from a_i, $\overline{b_i}$ 2. Invert all sign bits of x 3. Produce the 2's complement $-T_x(x)$, set the Nth bit
–(a–b)	$-[T_Y(y)+1]$	$-2^N + T_X(x)$	1. Obtain all x_i from a_i, $\overline{b_i}$ 2. Produce the 2's complement $T_x(x)$, set the Nth bit

$$d_i'' \begin{cases} S_i'' = M_i' \cdot \overline{\overline{S_{i-1}'}M_{i-1}'} = \overline{a_i \oplus b_i} \cdot (a_{i-1} + b_{i-1}) \\ M_i'' = M_i' \oplus \left(\overline{S_{i-1}'}M_{i-1}'\right) = \overline{a_i \oplus b_i} \cdot \overline{a_{i-1} + b_{i-1}} \end{cases} \quad 2 \le i \le N-1 \tag{6}$$

$$d_N'' \begin{cases} S_N'' = 0 \\ M_N'' = \overline{S_{N-1}'}M_{N-1}' = \overline{a_{N-1} + b_{N-1}} \end{cases} . \tag{7}$$

Each digit at the output d_i'' is expressed in sign-magnitude form and is a function of the signed-binary input or the input two's-complement operands, *a* and *b*. With these expressions, the signed-binary number representation (SBNR) of $T_x(x)+1$ is achieved in two gate delays. This number is converted to two's-complement or, alternatively, may be inverted while in signed-binary form to alternate between the *(a+b)* and *–(a+b)* functions.

Note that the conversion expressed by (3) cannot be directly implemented by an N-bit adder with set carry-in bit $c_0 = 1$, because $T_x(x)+1$ is not achieved. Although the conversion is similar to a regular addition, the function of the carry-in bit is different since the outputs are inverted.

Similar expressions are considered for the difference $a - b$. In this case, the SB digits x_i are produced from the initial sum y_i, which is obtained from the bit-wise sum $a_i + \overline{b_i}$. The function is equal to

$$(a-b) = T_y[y] + 1 = T_y[tr_{xy}(x)] + 1 = 1 + \sum_{i=0}^{N}(1 - x_i)2^i =$$

$$= 1 + [2^N - 1 - T_x(x)] = 2^N - T_x(x) \quad . \tag{8}$$

The *x* number is in signed-binary representation. The inverse $-T_x(x)$, the same number with toggled sign bits, is converted to TC to achieve $a - b$. The corresponding inverse function *–(a–b)* is

$$-(a-b) = -\left(2^N - T_x(x)\right) = -2^N + T_x(x) . \tag{9}$$

The implementation of the two difference functions is similar to that of the summation functions, with the exception of the +1 addition. This addition is conveniently incorporated in a prelogic stage by (4) - (7). The transformations and operations required to compute the four functions, with two alternative implementations of the *a+b* addition, are summarized in Table 1. Note that although the sign-magnitude combination "10" is forbidden, the sign inversion does not cause problems in the CLA performing the SB→TC conversion. This behavior occurs because in the case of this forbidden bit pair, the corresponding generate-propagate (*G-P*) signals of the CLA become "11" and the value of the generate bit does not affect the output result [9].

3. The Sign Bit

The summation of two N-bit numbers is generally an (N+1)-bit number. The results derived in the previous sections, however, are valid only if the input operands are N-bit positive numbers or if the output is N-bit with no overflow. Special care is required to set the Nth sign bit in order to achieve correct results for all cases. An alternative is to limit the input operands such that the output is constrained within the range of an N-bit TC number. In order to resolve this issue, the two's-complement addition of two N-bit numbers *a*+*b* is presented as

a	+	a_{N-1}	a_{N-2}	…	a_2	a_1	a_0
b		b_{N-1}	b_{N-2}	…	b_2	b_1	b_0
Partial positive sum of bits 0 : (N-2)		c_{N-1}	c_{N-2}	…	c_2	c_1	c_0
Actual summation result	r_N	r_{N-1}	r_{N-2}	…	r_2	r_1	r_0

The sum of all positive bits (N-2):0 is denoted by the N-bit number *c* and the final addition result by the (N+1)-bit number *r*. The lower N-1 bits of *c* and *r* are equal such that only the two most significant bits require attention.

The input sign bits a_{N-1} and b_{N-1} are both weighted by -2^{N-1}, while the output sign bit r_N has a weight of 2^N. The relations between all sign bits are listed in Table 2, where the following expressions are considered,

$$r_{N-1} = a_{N-1} \oplus b_{N-1} \oplus c_{N-1} = P_{N-1} \oplus c_{N-1} \quad , \qquad (10)$$

$$r_N = a_{N-1}b_{N-1} + (a_{N-1} \oplus b_{N-1})\overline{c_{N-1}} = G_{N-1} + P_{N-1}\overline{c_{N-1}} \quad . \qquad (11)$$

Table 2: Sign bit relations

Inputs			Result		
a_{N-1}	b_{N-1}	c_{N-1}	r_N	r_{N-1}	value
0	0	0	0	0	0
0	0	1	0	1	$+2^{N-1}$
0	1	0	1	1	-2^{N-1}
0	1	1	0	0	0
1	0	0	1	1	-2^{N-1}
1	0	1	0	0	0
1	1	0	1	0	-2^N
1	1	1	1	1	-2^{N-1}

G_i and P_i are the carry generate and carry propagate inputs of the CLA, which performs the SB→TC conversion. Note that the N-1st bit is the same bit as computed by the carry-lookahead adder. Only the Nth bit is changed and is similar to the carry c_N, with inverted $\overline{c}_{N-1}$ to account for the negative weight of the sign bits [9]. This function is achieved inside the adder by inverting the propagated carry c_{N-1} when c_N is computed. This result is correct for all four functions. Since the sign bit is controlled by (10) and (11), computing the Nth digit from (7) is unnecessary (only r_N and r_{N+1} are affected); therefore, only an N-digit SB number is required.

4. Numeric Examples

Examples are considered in this section and shown in Table 3 to verify the analytical results and illustrate the general ideas behind the conversion process. These examples do not consider a broad range of values, but rather are intended to demonstrate the sequence of operations required to compute the target functions.

The four functions are divided into two pairs. Each pair is implemented with a minor difference: a single conditional inversion of the sign bits in the signed-binary representation. The SB number for the two functions in each pair is represented by either $T_x(x)$ or $T_x(x)+1$. The sign signals are either inverted or not, and the SB number is mapped to the *G-P* inputs of a carry-lookahead adder. The adder outputs are inverted and the two's-complement sign bit is set according to (11) to achieve the final function.

Analogies with bit processing exist in each column and each row. Those similarities are discussed in more detail in section 6.

5. Alternative Transformations

Alternative relations between the sets $S_y(y)$ and $S_x(x)$ are discussed in this section. The transformation $tr_{xy} = 1 - z_i$ is denoted as T^0 in the following discussion. Rather than a strict requirement for duality (for the transform to be self-inverting), only a one-to-one mapping condition is imposed. One such transformation is

$$T^1: \quad y_i = x_i + 1 \quad : \quad -1 \rightarrow 0\ ;\ 0 \rightarrow 1\ ;\ 1 \rightarrow 2 \quad . \qquad (12)$$

In this case, the two's-complement of the *initial sum y* is

$$a + b = T_y(y) = T_y[tr_{xy}(x)] = \sum_{i=0}^{N-1} tr_{xy}(x_i)2^i = \sum_{i=0}^{N-1}(1 + x_i)2^i =$$

$$= 2^N - 1 + T_x(x) = 2^N + \overline{-T_x(x)} \quad . \qquad (13)$$

As in the previous transformation, relations for the other functions are based on T^1 and are listed in Table 4.

$$-(a+b) = -2^N + 1 - T_x(x) \quad , \qquad (14)$$

$$(a-b) = T_y[y] + 1 = T_y[T^1{}_{xy}(x)] + 1 = 1 + \sum_{i=0}^{N-1}\left[T^1{}_{xy}(x_i)\right] \cdot 2^i =$$

$$= 1 + \sum_{i=0}^{N-1}\left[(1 + x_i)\right]2^i = 1 + 2^N - 1 + T_x(x) = 2^N + T_x(x)\ , \qquad (15)$$

$$-(a-b) = -2^N - T_x(x) \qquad (16)$$

Table 4: Comparison of transformations

Arithmetic function	T^0: $y_i=1-x_i$	T^1: $y_i=1+x_i$
a + b	$2^N + \overline{T_X(x)}$	$2^N + \overline{-T_X(x)}$
	$2^N - (1 + T_X(x))$	$2^N - (1 - T_X(x))$
– (a + b)	$-2^N + 1 + T_X(x)$	$-2^N + 1 - T_X(x)$
a – b	$2^N - T_X(x)$	$2^N + T_X(x)$
– (a – b)	$-2^N + T_X(x)$	$-2^N - T_X(x)$

Note that either *(a–b)* or *–(a–b)* can be achieved with the same addition circuit, changing only the preprocessing step which maps a_i and b_i onto the x_i digits. The results are the same up to the Nth bit (which is the sign bit). The sign bit is controlled separately or ignored if overflow precautions are applied.

The remaining four of the six possible mappings from S_y to S_x are listed in Table 5. The relationships between S_Y and the SB sets S_{x1} and S_{x2} provide a means for fast digit-processing and efficient computation. Alternatively, transforms 3 to 6 require a more elaborate transformation to/from S_Y and can be useful for decomposing an initial sum number from S_y into two signed-binary numbers. Both transformations, T^0 and T^1, produce all of the functions listed in Table 4. These transformations may also be used interchangeably to switch between two functions by only changing the prelogic mapping stage.

Table 3: Numeric examples for the proposed transformations that compute the four functions, *(a+b)*, *–(a+b)*, *(a–b)*, and *–(a–b)*

Elementary operations		Example I						Example II					
		Sum ± (a+b)			Difference ± (a–b)			Sum ± (a+b)			Difference ± (a–b)		
A			10011011	–101		10011011			10101110	-82		10101110	
b ($\bar{b}$)			11101001	–23		00010110			00111100	62		11000001	
SB	$Sign_i$	$T_x(x)+1$	010000100		$T_x(x)$	00010010		$T_x(x)+1$	001101100		$T_x(x)$	10000000	
	$Magn_i$		010000100			01110010			011101100			10010000	
G-P	Invert signs, $G_i = \overline{Sign_i}$		101111011			11101101			110010011			01111111	
	$P_i = \overline{Magn_i}$		101111011			10001101			100010011			01101111	
Convert to TC (CLA)			01 01111011			101001101			1000010011			010001111	
Invert all bits (N-1)-0 Set N^th bit			1 10000100	–124		1 10110010	–78		1 111101100	-20		1 01110000	–144
Inverted functions													
G-P	Skip sign invertion $G_i = Sign_i$		010000100			00010010			001101100			10000000	
	$P_i = \overline{Magn_i}$		101111011			10001101			100010011			01101111	
Convert to TC (CLA)			1010000011			010110001			0111101011			101101111	
Invert all bits (N-1)-0 Set N^th bit			0 01111100	124		0 01001110	78		0 000010100	20		0 10010000	144

Table 5: All possible mappings of intermediate results

Input bits	Initial sum	$x_i=1-y_i$	$x_i=1+y_i$	$x_{1i}=x_{3i}+x_{4i}$		$x_{2i}=x_{5i}+x_{6i}$	
	S_Y	S_{x1}	S_{x2}	S_{x3}	S_{x4}	S_{x5}	S_{x6}
00	0	1	$\bar{1}$	1	0	$\bar{1}$	0
01, 10	1	0	0	$\bar{1}$	1	1	$\bar{1}$
11	2	$\bar{1}$	1	0	$\bar{1}$	0	1

6. Example application for a CDMA Pseudonoise Scrambler

Modern CDMA cellular systems employ spread spectrum technology to provide multiuser access. An integral part of the transceiver is the scrambling operation, which involves the multiplication of the chip sequence with a pseudonoise (PN) code in order to distinguish signals from asynchronous users. In the UMTS third generation wireless standard, the scrambling code is complex, thereby requiring a complex multiplication [3]. Since the components of the complex PN code take binary values in the set $\{-1, +1\}$, the scrambling multiplier can be optimized. The transformations described in the previous sections are applied to an efficient implementation of the scrambler block in a CDMA wireless receiver.

The function of the scrambler is the multiplication of a complex input signal $a + jb$ by the PN code $PN_{re} + jPN_{im}$, where PN_{re} and PN_{im} are in the binary set of $\{-1,+1\}$. The complex output signal is $A + jB = (a + jb)\cdot(PN_{re} + jPN_{im})$. Note that the real and imaginary components of the output signal take one of the four values described in Table 6, controlled by the pseudonoise code signals. The four functions, $(a+b)$, $-(a+b)$, $(a-b)$, and $-(a-b)$, are required to implement a scrambler. Area, power, and speed resources are saved if the circuit realization is accomplished with conditional switching of a minimum number of logic elements by the PN signals along the critical path.

The efficient realization of these four functions has been described in previous sections and numeric examples are presented in section 4. It is evident from Table 3 that there is only one difference in the methods of computing the sum functions, $(a+b)$ and $-(a+b)$, and the difference functions $(a-b)$ and $-(a-b)$. These functions only differ by the initial mapping of the input operands a and b to the signed-binary number. As discussed in sections 3 and 4, the SB number $T_x(x)+1$ is required to produce the sum functions, while $T_x(x)$ is required for the difference functions. From a different perspective, both of the direct functions, $(a+b)$ and $(a-b)$, differ from the inverse functions, $-(a+b)$ and $-(a-b)$, respectively, by a single step - the inversion of all sign bits of the SB representation.

Table 6: Input/output relations for a ±1 complex multiplier

PN code		Outputs	
PN_{re}	PN_{im}	A	B
1	1	a – b	a + b
1	–1	a + b	– (a – b)
–1	1	– (a + b)	a – b
–1	–1	– (a – b)	– (a + b)

Based on these observations, an architectural solution for the complex ±1 multiplier is proposed in [3, 4]. There are two distinct branches. Two of the four functions are implemented along each of these branches. As shown in Table 6, for any PN code, one summation function and one difference function are computed. The delay of the critical path is reduced by approximately 30% in the proposed signed-binary architecture as compared to conventional realizations [3, 4]. This enhanced speed is realized by reducing the number of carry propagation chains in the proposed architecture.

7. Conclusions

An analytical treatment of number representations for efficient VLSI arithmetic circuits is presented. It is shown that a variety of arithmetic functions, $(a+b)$, $-(a+b)$, $(a-b)$, and $-(a-b)$, can be realized with significant resource savings. Alternative transformations and potential applications are also described and examples are presented to support the analytic results. An application of the proposed transformations in a wireless CDMA scrambler is discussed where a significant speed benefit as compared to conventional techniques is achieved.

8. References

[1] M. Soderstrand, W. Jenkins, G. Jullien, and F. Taylor, *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing*, IEEE Press, 1986.

[2] T. Stouraitis and V. Paliouras, “Considering the Alternatives in Low-Power Design,” *IEEE Circuits and Devices*, Vol. 17, No.4, pp. 22-29, July 2001.

[3] B. Andreev, E. G. Friedman, and E. L. Titlebaum, "Efficient Implementation of a Complex ±1 Multiplier," *Proc. of the ACM Great Lakes Symp. on VLSI*, pp. 83-88, April 2002.

[4] B. D. Andreev, E. L. Titlebaum, and E. G. Friedman, "Complex ±1 Multiplier Based on Signed-Binary Transformations," *Journal of VLSI Signal Processing*, 2003.

[5] G. M. Blair, "The Equivalence of Twos-complement Addition and the Conversion of Redundant-binary to Twos-complement Numbers," *IEEE Transactions on Circuits and Systems I*, Vol. 45, No. 6, pp. 669-671, June 1998.

[6] H. Srinivas and K. Parhi, "A Fast VLSI Adder Architecture," *IEEE Journal on Solid-State Circuits*, Vol. 27, No. 5, pp. 761-767, May 1992.

[7] A. Gonzalez and P. Mazumder, "Redundant Arithmetic, Algorithms and Implementations," *Integration, The VLSI Journal*, Vol. 30, No. 1, pp. 13-53, November 2000.

[8] J. M. Dobson and G. M. Blair, "Fast Two's Complement VLSI Adder Design," *Electronic Letters*, Vol. 31, No. 20, pp. 1721-1722, September 28, 1995.

[9] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, 1993.