On the Effectiveness of Residue Code Checking for Parallel Two's Complement Multipliers**

U. Sparmann*,‡ and S.M. Reddy†
* Computer Science Department, University of Saarland, D 66041 Saarbrücken, Germany
† Department of ECE, University of Iowa, Iowa City, Iowa 52242, USA

Abstract

The effectiveness of residue code checking for on-line error detection in parallel two's complement multipliers has up to now only been evaluated experimentally for few architectures. In this paper a formal analysis is given for most of the current multiplication schemes. Based on this analysis it is shown which check bases are appropriate, and how the original scheme has to be extended for complete error detection at the input registers and Booth recoding circuitry. In addition, we argue that the hardware overhead for checking can be reduced by approximately one half if a small latency in error detection is acceptable. Schemes for structuring the checking logic in order to guarantee it to be self-testing, and thus achieve the totally self-checking goal for the overall circuit, are also derived.

Keywords: Self-checking circuits, fault-secure, self-testing, residue codes, parallel two's complement multiplication.

**This work is an extended version of a paper presented at the 24th Int. Symposium on Fault-Tolerant Computing, Austin, Texas, June 15-17, 1994.

‡Work done while visiting the University of Iowa, supported by DFG, Grant No. Sp431/1-1 and SFB 124 – 'VLSI Entwurfsmethoden und Parallelität'.

1 Introduction

With increasing level of integration the realization of more and more complex and fast parallel algorithms as VLSI circuits is feasible. At the same time, with shrinking device geometries the susceptibility to physical disturbances rises, and system reliability becomes a serious problem. This is especially true for critical computer applications where huge sums of money, or even human lives are at stake. Thus, in order to bring the benefits of VLSI to more and more aspects of our daily lives, it is crucial to develop methods for detecting permanent and transient errors *on-line*, i.e. during the normal operation of the system, before they can result in any harm.

One of the most important parallel algorithms which is part of nearly every computer system is fast multiplication. Methods for on-line error detection in parallel multipliers by coding techniques have been studied considerably in the literature: parity prediction in [1, 2], two and three rail encoding in [3, 4], Berger codes in [5] and residue codes in [6, 7]. The overhead induced by the first three approaches, i.e. parity prediction, two and three rail encoding, and Berger codes, is proportional to the overall gate complexity of the multiplier which is of order n^2 for an n-bit parallel multiplier. The only coding technique which achieves an overhead proportional to n, and thus becomes more and more attractive with increasing operand size compared to the other schemes, is residue code checking.[1] Industrial applications of residue codes for checking parallel multipliers have been reported for example in [10, 11].

Up to now, analytical studies concerning the effectiveness of residue code checking for on-line error detection in multipliers have mostly focused on bit serial multiplication techniques [12, 13]. For parallel multiplication only simulation based analysis for specific architectures has been reported [10, 6, 7], which indicates that a small percentage of errors goes undetected. (As an example, in [7] an average probability of 0.052 for not detecting an error has been measured for a 12-by-12-bit two's complement multiplier with respect to modulo 3 checking. But note, that in this work the sign bit of the result has not been checked appropriately.) The problem with the above results is that they are only valid for specific architectures and operand lengths. More severely, they do not analyze the reasons for undetected errors, nor do they show how to achieve complete error detection by appropriate choice of the check base and (or) enhancements to the original checking scheme.

In contrast to the above approaches, the study of this paper is done analytically. We exactly characterize the dependency between the faults inside the multiplier and the possible error values which can be caused by them at the primary outputs. Because of the similarities in their algorithmic principles this analysis can be carried out for several different multiplier realizations [14, 15, 16, 17, 18]. From the analysis it follows that 7 is the smallest check base to achieve complete error detection. For check base 3, which is very popular in practice since it implies the least overhead, we argue that its effectiveness depends on the realization of the basic cells. Regarding the input registers and Booth recoding logic we show that the original checking scheme is not sufficient, but has to be enhanced by additional checks. Since these additional checks are expensive in terms of hardware overhead, we analyze their necessity carefully, and prove that they can be omitted if a small latency in error detection (less than 64 operations on the average) is acceptable. In order for a circuit to be totally self-checking, also the self-testing property has to be guaranteed. It turns out that this property is not achieved by standard designs for the checkers monitoring the output of the Booth recoder and the overall multiplier. We will show how to solve this problem by appropriately restructuring the checkers. Finally, we give estimates for the hardware overhead in different architectures. The estimates indicate that on-line error detection by residue codes is very cost effective for large parallel multipliers.

The paper is organized as follows. Section 2 gives an overview of parallel multiplier architectures, introduces some basic definitions concerning self-checking circuits, and illustrates the main problems by examples. Fault effects are studied in Section 3, where it is also shown that 7 is the smallest check base to achieve complete error detection. In Section 4 we discuss input registers and Booth recoded multipliers, give the necessary enhancements to the original checking scheme, and show that these can be omitted if a small latency in error detection is acceptable. Realization of the checking logic in order to guarantee the self-testing property is discussed in Section 5. Section 6 gives estimates for the hardware overhead in different architectures. Finally, conclusions are given in Section 7.

[1] If time redundancy is admissible, hardware overhead $O(n)$ can also be achieved by time redundant techniques like recomputing with shifted operands [8], or recomputing with duplication with comparison [9].

2 Preliminaries

This section explains the basic algorithmic principles of todays parallel two's complement multipliers, reviews some definitions concerning self-checking circuits, and illustrates the main problems attacked in this paper by examples.

We start by giving some notations used throughout the text: The set $\{0,1\}$ of binary values is denoted by $\mathbf{B}$. $\mathbf{N}$ ($\mathbf{Z}$) denotes the set $\{1,2,3,\ldots\}$ ($\{\ldots,-2,-1,0,1,2,\ldots\}$) of all natural numbers excluding zero (all integers), and $\mathbf{N}_0 := \mathbf{N} \cup \{0\}$. For $i,j \in \mathbf{Z}$, $i \leq j$, we define $[i:j] := \{k \in \mathbf{Z} | i \leq k \leq j\}$ and $]i:j[:= \{k \in \mathbf{Z} | i < k < j\}$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are two sets then $\mathcal{A}_1 \setminus \mathcal{A}_2 := \{a \in \mathcal{A}_1 | a \notin \mathcal{A}_2\}$ denotes the set difference of $\mathcal{A}_1$ and $\mathcal{A}_2$.

2.1 VLSI implementations of parallel multiplication

In the following we identify an α-bit binary string $z = z_{\alpha-1}\ldots z_0 \in \mathbf{B}^\alpha$ with its interpretation as an unsigned binary number $\sum_{i=0}^{\alpha-1} z_i 2^i$. Whatever is meant will become clear from the context. The interpretation of z as a two's complement number will be denoted by $I(z) := -z_{\alpha-1}2^{\alpha-1} + \sum_{i=0}^{\alpha-2} z_i 2^i$. A two's complement multiplier gets as inputs two n-bit numbers $x = x_{n-1}\ldots x_0$ (the multiplier) and $y = y_{n-1}\ldots y_0$ (the multiplicand) and computes the product $p = p_{2n-1}\ldots p_0$ such that $I(p) = I(x) \cdot I(y)$.

Typically, parallel multipliers proceed in the following three steps: (1) Partial product generation (PPG). (2) Reduction of the partial products to two numbers with their sum modulo 2^{2n} equal to p. (3) Addition of these two numbers by a carry propagate adder.

In this paper we will concentrate on the first two steps. Methodologies to design efficient carry propagate adders such that they can be checked with minimum check bases can be found for example in [19, 20, 21, 22].

Let us first review some well known facts concerning the realization of step (2). Partial product reduction (PPR) is normally done by trees of full and half adders (FAs and HAs). Each tree reduces the bits of one column of the partial product matrix (PPM) and the carry bits from the previous tree to one or two output bits and the carry bits to the next column. The structure of these column trees can be chosen individually for each column in order to minimize the overall delay of the circuit [15, 17]. To obtain more regular VLSI layouts FAs and HAs are often grouped into carry save adders (CSAs) and the summation of the partial products is done by a tree of CSAs in a row oriented manner [18]. There is a great variety of possibilities for structuring such a CSA tree [18]. The simplest scheme, well suited for pipelining, is the linear tree. For fast multiplication Wallace trees [23], or related more regular structures [24, 25] are usually applied.

Let us now concentrate on step (1), i.e. partial product generation. For simplicity of presentation we defer the treatment of Booth recoding [18] for PPG to Section 4.

Full sign extension The partial products $pp_0,\ldots,pp_{n-1}$ are generated in two's complement representation, i.e. $I(pp_0) = x_0 2^0 \cdot I(y),\ldots,I(pp_{n-2}) = x_{n-2}2^{n-2}\cdot I(y)$, $I(pp_{n-1}) = -x_{n-1}2^{n-1}\cdot I(y)$. They are obtained by the following rules [18]:
pp_i, $i \in [0:n-2]$, is generated by 'anding' the bits of y with x_i, adding i trailing zeros and sign extending up to bit position $2n$, i.e.[2]

$$pp_i = [(x_i y_{n-1})^{n-i}, x_i y_{n-1}, \ldots, x_i y_0, 0^i].$$

(z^α is short for $z\ldots z$ α-times, and $x_i y_j := x_i \wedge y_j$.)
For pp_{n-1} we additionally have to complement the bits of y and add x_{n-1} in bit position n modulo 2^{2n} to obtain the representation of $(-x_{n-1}\cdot I(y))\cdot 2^{n-1}$, i.e.

$$\begin{aligned} pp_{n-1} = ([x_{n-1}\overline{y_{n-1}}, x_{n-1}\overline{y_{n-1}}, \ldots, x_{n-1}\overline{y_0}, 0^{n-1}] \\ + [0^n, x_{n-1}, 0^{n-1}]) \bmod 2^{2n}. \end{aligned}$$

As an example, Figure 1 shows the PPM for a 5-by-5 bit multiplication with $x = 10101$ ($I(x) = -11$), $y = 01101$ ($I(y) = 13$) and $p = 1101110001$ ($I(p) = -143$). For the partial products trailing zeros have been omitted, since they do not have to be considered during the addition process. Figure 2 shows a schematic diagram of the PPM. The sign bits are indicated by dark bullets. For simplicity we use the same symbols for all rows despite the fact that the entries in the last two rows are computed in a slightly different manner.

0	0	0	0	0	0	1	1	0	1
0	0	0	0	0	0	0	0	0	
0	0	0	0	1	1	0	1		
0	0	0	0	0	0	0			
1	1	0	0	1	0				
					1				
1	1	0	1	1	1	0	0	0	1

Figure 1: PPM for $[10101]\cdot[01101]$

[2] In the following we will often enclose a binary string into brackets in order to distinguish it more clearly from the text.

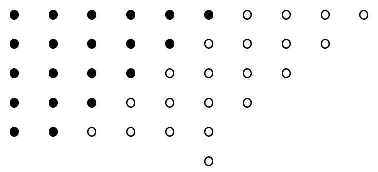

Figure 2: PPM with sign extensions

Figure 3 shows a possible realization for the PPG and PPR part. The cells of the PPG part are indicated by circles, corresponding to the schematic PPM of Figure 2. For simplicity, cell inputs have been omitted. The PPs are reduced to two numbers by a linear tree of three CSA adders. Each CSA consists of FAs and the upper two ones in addition include a trailing HA. Since the addition is in two's complement it has to be done modulo 2^{2n}, i.e. the outgoing carries of the CSAs are cut off. The first CSA reduces pp_0, pp_1 and pp_2 to two numbers with their sum modulo 2^{2n} equal to $(pp_0 + pp_1 + pp_2) \bmod 2^{2n}$. The second CSA then gets as input these two numbers and pp_3, and so on.

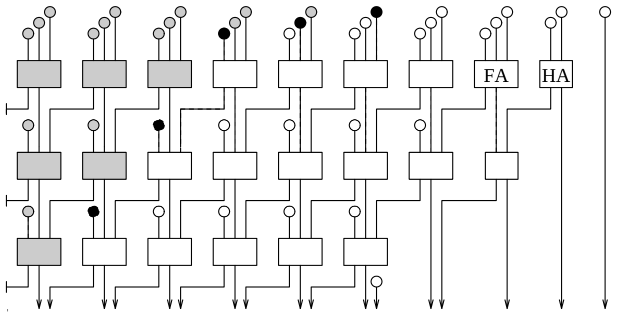

Figure 3: PPG and PPR with full sign extension

There are many approaches to eliminate the sign extensions in the upper left triangle of the PPM, thus saving the grey shaded cells in Figure 3. In the following we discuss the two approaches most suitable for VLSI implementation.

Addition of 'correction terms' The most well known representative of this scheme is the Baugh-Wooley multiplier [14]. A slightly simpler method is given in [18]. Since the scheme of [18] is better suited for VLSI implementation and generalizes naturally to Booth recoded multipliers [18], we will restrict to it in the following. Our results concerning residue code checking are not depending on the specific correction terms and thus are also valid for the Baugh-Wooley scheme.

The partial product matrix for the scheme of [18] differs from the complete partial product matrix with sign extension in the following aspects: The sign bit of each partial product is inverted and instead of the sign extension bits two additional ones are added in the n-th and the $2n$-th column. The PPM with correction terms corresponding to the multiplication example of Figure 1 is depicted in Figure 4. The corresponding schematic diagram is given in Figure 5.

					1				
					1	1	1	0	1
				1	0	0	0	0	
			1	1	1	0	1		
		1	0	0	0	0			
	0	0	0	1	0				
1					1				
1	1	0	1	1	1	0	0	0	1

Figure 4: PPM for $[10101] \cdot [01101]$

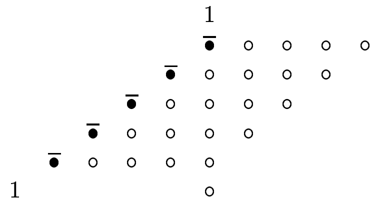

Figure 5: PPM with correction terms

For full sign extension as well as correction term addition, the PPR can be realized by FA and HA trees structured in an arbitrary manner, i.e. either column oriented or as CSA trees. The following scheme is only applicable if PPR is realized by CSA trees.

'CSA folding' or 'delayed sign extension' Consider again the sign extended scheme of Figure 3. Clearly, the PPG cells computing the sign extensions of pp_0, pp_1 and pp_2 can be replaced by one cell each (colored black in Figure 3) with appropriate fan-out of the output. Since now the four FAs on the left of the upper CSA all receive the same inputs they can be replaced by one FA, i.e. the three shaded FAs in Figure 3 can be omitted. Now it can be seen that the second CSA has three FAs to its left which receive as input the same values and thus can be replaced by just one FA and its associated PPG cell. Repeating this process also for the lower CSA we finally arrive at the much cheaper design of Figure 6 [16] in which all the shaded cells of Figure 3 have been eliminated.

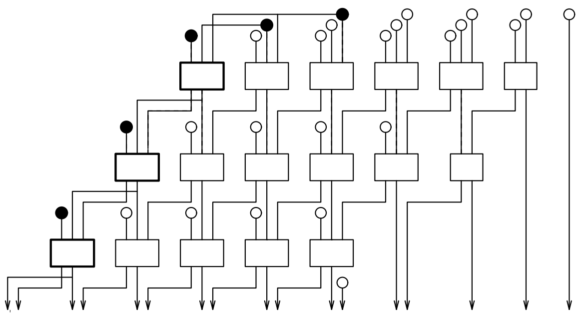

Figure 6: PPG and PPR with folded CSAs

For this scheme the partial products need no sign extension at all, since sign extension is 'delayed' to the actual addition.

2.2 Checking multipliers by residue codes

This section illustrates the main problems attacked in this paper by examples. In addition, it briefly reviews some notions from the theory of self-checking circuits and residue codes which will be of importance later. For further reference in this area, see for example [19, 26, 27, 28] for self-checking circuits, and in addition [29, 30, 31] for residue codes. Results on the effectiveness of residue codes for array dividers can be found in [32].

The inputs (outputs) of a self-checking circuit S are encoded in an error-detecting code $\mathcal{I}$ ($\mathcal{O}$). Let F be the set of most likely faults in S. To achieve fault secureness with respect to F, the code $\mathcal{O}$ has to be selected such that any erroneous output due to a fault from F can be detected by a code check.

Definition 1 *Let $f \in F$ and S^f denote the circuit S faulted by f. The output of S (S^f) for input i is given by $S(i)$ ($S^f(i)$).*

S is fault secure *for fault set $F \iff$ for any input $i \in \mathcal{I}$ and for any fault $f \in F$, $S(i) \neq S^f(i)$ implies $S^f(i) \notin \mathcal{O}$.*

Figure 7 shows the basic configuration of a residue code checked multiplier S. Here, M denotes an n-bit two's complement multiplier, $b \in \mathbf{N}$ is the *check base*, and M' denotes a multiplier modulo b. The residues $I(x) \bmod b$, $I(y) \bmod b$ and $(I(x) \cdot I(y)) \bmod b$ are represented as unsigned binary numbers. The output code space is $\mathcal{O} = \{(p, I(p) \bmod b) | p \in \mathbf{B}^{2n}\}$. An erroneous output p^f of the multiplier M is detected if and only if:

$$(p^f, I(p) \bmod b) \notin \mathcal{O}$$

$$\begin{aligned} &\iff I(p) \bmod b \neq I(p^f) \bmod b \\ &\iff |I(p) - I(p^f)| \bmod b \neq 0. \qquad (1) \end{aligned}$$

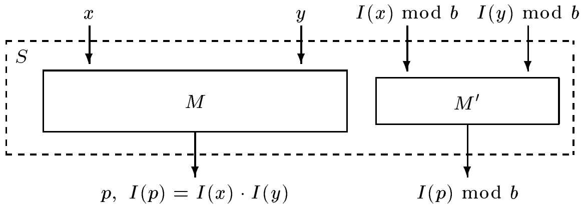

Figure 7: Multiplier checked by residue code

Note that adders are often checked by a code of the form $(z, z \bmod b)$. Thus, a code transformation may be necessary for a system consisting of adder and multiplier. This transformation from $(z, I(z) \bmod b)$ to $(z, z \bmod b)$ or vice versa can be computed with low hardware cost by applying the relation $z = I(z) + z_{\alpha-1}2^{\alpha}$.

Let us now fix the **fault model** for the residue code checked multiplier as shown in Figure 7. For its fault set F we assume that exactly one module M or M' is defective. In M' arbitrary faults are admitted. In M we restrict to (single) *cellular faults*, i.e. we assume that exactly one basic cell, i.e. PPG cell, FA or HA, of M is faulty. A fault is allowed to modify the cells functional behavior in an arbitrary manner.

Let $E(M)$ denote the set of absolute error values with respect to two's complement interpretation resulting from cellular faults in M, i.e.[3]

$$\begin{aligned} E(M) := \{&|I(p) - I(p^f)| \,|\, f \text{ cellular fault in } M, \\ &p = M(x,y),\, p^f = M^f(x,y),\, x,y \in \mathbf{B}^n\} \setminus \{0\}. \end{aligned}$$

Clearly, any erroneous output of M' results in a noncode word. Based on Equation 1, we thus obtain [26, 27]:

Lemma 1 *S is fault secure for $F \iff$*

$$\forall e \in E(M) : e \bmod b \neq 0.$$

The most important point in residue code checking is the selection of the check base b. In order to achieve fault secureness with a minimum hardware overhead, b should be chosen minimum such that it fulfills Lemma 1. To do so, we first have to characterize the set $E(M)$ of absolute error values. One major concern in doing so will be the analysis of overflows caused by faulty cells. The following example shows

[3] $M(x,y)$ ($M^f(x,y)$) denotes the output of the correct (faulty) multiplier for input (x,y).

that due to an overflow a simple local error value ($\alpha 2^i$, $\alpha \in \{1,2,3\}$) can be transformed into a more difficult to check global error ($e = 2^{2n} - \alpha 2^i$).[4]

Example 1 *Figure 8 gives the fully sign extended PPM for the multiplication of* $x = y = [10\ldots0]$ $(I(x) = I(y) = -2^{n-1})$.

$$
\begin{array}{ccccccccc}
0 & 0 & 0 & \ldots & 0 & 0 & \ldots & 0 & 0 \\
0 & 0 & 0 & \ldots & 0 & 0 & \ldots & 0 & \\
 & & & \vdots & & & & & \\
0 & 0 & 0 & \ldots & 0 & 0 & & & \\
0 & 0 & 1 & \ldots & 1 & & & & \\
 & & & & 1 & & & & \\
\hline
0 & 1 & 0 & \ldots & 0 & 0 & \ldots & 0 & 0
\end{array}
$$

Figure 8: PPM for $[10\ldots0] \cdot [10\ldots0]$

Assume that one of the full adders summing up the zero entries of significance 2^{2n-3} *computes a faulty output* 11 *instead of the correct* 00. *In this case* $3 \cdot 2^{2n-3}$ *is added to the correct result* $p = [010\ldots0]$ $(I(p) = 2^{2n-2})$. *Since the maximum positive value representable with* $2n$ *bits in two's complement is* $2^{2n-1} - 1$, *a two's complement overflow results. As a consequence the faulty output* $p^f = (p + 3 \cdot 2^{2n-3}) \bmod 2^{2n} = [1010\ldots0]$ *is negative, i.e.* $I(p^f) = -(3 \cdot 2^{2n-3})$. *The corresponding absolute error value is* $|I(p) - I(p^f)| = 5 \cdot 2^{2n-3}$. *Thus, the local error value* $3 \cdot 2^{2n-3}$ *has resulted in a global error of* $5 \cdot 2^{2n-3} = 2^{2n} - (3 \cdot 2^{2n-3})$ *because of overflow.*

The relation between local and global error effects is studied in detail in Section 3 for all the multiplication schemes introduced in Section 2.1. Especially, it is shown there that because of 'redundancy in the number representation of the final product'[5] faulty cells can only cause product overflows in very rare cases. (The above example actually gives the 'extreme case' of what can happen.) As a result of the analysis we will obtain that for $b = 7$ all global error effects can be detected.

If we consider a complete system checked by residue codes then clearly, also faults in the input registers must be detected. In this case fault secureness can not be achieved at reasonable costs for the original checking scheme as illustrated by the following example.

[4] The error values of $\{\alpha 2^i | i \in [0 : 2n-1],\ \alpha \in \{1,2,3\}\}$ can be detected with a constant check base b independent of n, e.g. $b = 5$ or $b = 7$. If in addition all error values of $\{2^{2n} - 2^i | i \in [0 : 2n-1]\}$ have to be detected, it can be shown [33] that the check base must increase linearly with the product length, i.e. $b > 2n$.

[5] From the range $[-2^{2n-1} : 2^{2n-1} - 1]$ representable with $2n$ bits in two's complement only the numbers from $[-2^{n-1} \cdot (2^{n-1} - 1) : 2^{2n-2}]$ actually occur during normal operation.

Example 2 *Assume that the register cell computing the least significant* x*-input bit* x_0 *is faulty and produces output* 1 *instead of the correct value* 0. *Then* $e = |I(x) \cdot I(y) - (I(x)+1) \cdot I(y)| = |I(y)| \in [0 : 2^{n-1}]$. *Thus, by Lemma 1, we must choose* $b > 2^{n-1}$ *which is clearly too expensive.*

A similar problem arises in Booth recoded multiplication for the logic recoding the x-operand. A solution to these problems is given in Section 4, it consists in adding local checks for the input registers (recoding logic) to the basic scheme of Figure 7. These additional checks are rather expensive, i.e. they nearly constitute half of the checking overhead. Thus, we will analyze the necessity of these checks carefully. As a result, we will obtain that they may be omitted if a small latency in error detection (only 64 operations on the average) can be tolerated.

Up to now we only considered fault secureness which guarantees that any testable fault from the fault set F will be detected. In order to achieve the *totally self-checking* goal, we also have to make sure that the occurrence of an untestable fault does not destroy the fault secure property. This can be done by guaranteeing that the circuit is self-testing [26, 27].

Definition 2 *Let* S *be a self-checking circuit with input code space* $\mathcal{I}$, *output code space* $\mathcal{O}$. *The set of all* normal inputs *to* S, *i.e. all code words which actually occur as circuit inputs during normal (fault free) operation, is denoted by* $N \subset \mathcal{I}$.

(a) *S is called* self-testing *for a fault* $f \in F \iff$ *there exists an* $i \in N$ *such that* $S^f(i) \notin \mathcal{O}$.

(b) *S is called* self-testing *for fault set* $F \iff S$ *is self-testing for every fault from* F.

For the residue code checked multiplier, as given in Figure 7, in most applications $N = \mathcal{I}$ will be fulfilled[6], i.e. arbitrary inputs can be applied to M and M'. Thus, if M and M' are realized without internal redundancy, then fault secureness automatically implies the self-testing property.

For the code checker monitoring the outputs of the self-checking multiplier, the self-testing property can not be guaranteed by simply removing redundancy, as illustrated by the following example.

[6] Otherwise, we may add a built-in self-test to the self-checking module. This can be done at very low cost, since, because of the on-line error detection logic, no output compression circuit is needed.

Example 3 *A block diagram of the residue checker monitoring the multiplier outputs is given in Figure 9 [34]. Here,* $\mathrm{mod}_{TC}\ b$ *denotes a circuit computing the residue of its input in two's complement representation, i.e.* $I(p) \bmod b$. *The two-rail checker (2-R-check) compares its two input operands and indicates an error iff they are not inverse to each other.*

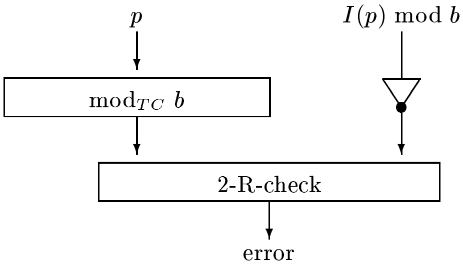

Figure 9: Residue checker

Since for an n-bit multiplier $I(x), I(y) \in [-2^{n-1} : 2^{n-1}-1]$, *we conclude that for the fault free product* $I(p) \in R' := [-2^{n-1}\cdot(2^{n-1}-1) : 2^{2n-2}]$. *Thus, from the total range* $R := [-2^{2n-1} : 2^{2n-1}-1]$ *of numbers representable with* $2n$ *bits in two's complement only less than one half actually occurs during normal operation.*

If we apply redundancy removal techniques to eliminate the circuitry inside $\mathrm{mod}_{TC}\ b$ *which is only testable by inputs from* $R \setminus R'$, *the modified circuit will compute wrong residues for some inputs from the range* $R \setminus R'$. *But this implies that the residue checker would not be able to detect some noncode words, and an erroneous multiplier output can go undetected.*

A similar situation also occurs for the checking circuitry needed for the Booth recoder. This is due to the fact that this circuit recodes an n-bit number to a signed digit representation of length $\frac{3}{2}n$. Thus, also for the Booth recoder only a small subset of all possible output combinations are applicable during fault free operation. Section 5 shows how to chose the structure of residue and two-rail checkers in order to guarantee the self-testing property in spite of the above problems.

3 Error analysis and check base selection

In this section we will focus on the internal logic of the multiplier M, register faults are discussed in the next section. We proceed in three steps: First, we prove that the fault effects at the primary outputs are 'equivalent' for all the architectures introduced in Section 2.1 (Lemma 2). These fault effects are then interpreted in two's complement representation to obtain the set $E(M)$ (Lemma 3). Finally, check base selection can be done by a simple application of Lemma 1 (Theorem 1).

We first consider the *PPG and PPR part* of M. Let us assume that these parts are built according to the *first or second scheme* of Section 2.1, CSA folding will be considered later. A faulty full or half adder in column $i+1$ of the PPR part can cause a local error value of the form $\{-\alpha 2^i, \alpha 2^i\}$, where $\alpha \in [1:3]$. If a PPG cell in column $i+1$ computes a faulty output, this introduces a local error value of $\{-2^i, 2^i\}$ and thus, is a special case of the above errors. Let p (p^f), $p \neq p^f$, denote the correct (faulty) output of the multiplier faulted by a cellular fault in the above parts. Since summation of the partial products is done modulo 2^{2n}, we conclude that:

$$p^f = (p \pm \alpha 2^i) \bmod 2^{2n},\ \alpha \in [1:3], i \in [0:2n-1] \quad (2)$$

Consider now *CSA folding*. We compare a folded multiplier M (see Figure 6) with its corresponding sign extended architecture M_{se} (see Figure 3) in order to determine the global effect of local errors in M. Let C_{fol} (C) denote the set of folded (unfolded) cells in M, i.e. C_{fol} consists of the 'leftmost' cells of M (drawn in bold in Figure 6) and C comprises all remaining cells. For the cells of C and their connections there is a one to one correspondence between M and M_{se}. If one of them computes a faulty value the effect on the primary outputs of M and M_{se} is obviously the same, and thus of the form given in Equation 2. If a cell $c \in C_{fol}$ in column $i+1$ exhibits an error the global effect in M is the same as if all cells to the left of and including c in M_{se}, which have been identified with c in M, compute the same erroneous output. Thus, we obtain:

$$\begin{aligned} p^f &= (p \pm \alpha(2^i + 2^{i+1} + \ldots + 2^{2n-1})) \bmod 2^{2n} \\ &= (p \pm \alpha(2^{2n} - 2^i)) \bmod 2^{2n} \\ &= (p \mp \alpha 2^i) \bmod 2^{2n} \end{aligned}$$

Let us now look at the *carry propagate adder* (CPA). If this adder is realized as a carry ripple adder, then clearly the same argumentation used to derive Equation 2 is also valid for cellular faults in it. The delay of carry ripple adders is very high and thus, faster addition schemes are usually applied in parallel multipliers. In most designs carry select adders [35] are chosen, since they combine low area with high speed, and their structure can be easily adapted to the output arrival times of the PPR part [15, 17]. In [21] based on [36, 37] a family of adders (ADD_{DP}) has been characterized which is very powerful (including carry select and conditional sum [38] like structures as special cases) and at the same time only exhibits fault effects of the form given in Equation 2 with respect to cellular faults. Combining these results with the above observations for the PPG and PPR part we obtain:

Lemma 2 *Let M be an n-bit multiplier, with PPG and PPR part constructed according to one of the schemes in Section 2.1, and the CPA realized as a carry ripple adder or an arbitrary member of the family ADD_{DP} characterized in [21].*

For the correct (faulty) output p (p^f), $p \neq p^f$, of the multiplier due to a cellular fault it holds

$$p^f = (p + w) \bmod 2^{2n}$$

where $w \in \{\alpha 2^i, 2^{2n} - \alpha 2^i | \alpha \in [1:3],\ i \in [0:2n-2]\}$.

Proof: Basically, the lemma resembles Equation 2 with the following two differences: (1) Since $-v \equiv 2^{2n} - v \bmod 2^{2n}$, the subtraction in Equation 2 has been rewritten as an addition. (2) The terms for $i = 2n-1$ are a subset of those obtained for $i = 2n-2$ and thus, have been omitted. ■

In order to determine the absolute error value $e := |I(p) - I(p^f)|$ we have to reconsider the above lemma with respect to two's complement interpretation. Addition modulo 2^{2n} corresponds to the addition of two $2n$-bit two's complement numbers. If there is *no overflow* during this addition, we have $I(p^f) = I(p) + I(w)$ and thus $e = |I(w)|$. Since

$$I(w) \in \{\alpha 2^i, -\alpha 2^i | \alpha \in [1:3],\ i \in [0:2n-3]\} \cup \{-2^{2n-1}\},$$

we obtain:

$$e \in \{\alpha 2^i | \alpha \in [1:3],\ i \in [0:2n-3]\} \cup \{2^{2n-1}\}.$$

Now consider the case of *overflow*, i.e. $I(p) + I(w) \notin [-2^{2n-1} : 2^{2n-1} - 1]$. It can be easily shown that in two's complement:

$$I(p^f) = \begin{cases} (I(p) + I(w)) - 2^{2n} & \text{for pos. overflow} \\ (I(p) + I(w)) + 2^{2n} & \text{for neg. overflow} \end{cases}$$

Since $I(x), I(y) \in [-2^{n-1} : 2^{n-1} - 1]$ we know that $I(p) \in\ ]-2^{2n-2} : 2^{2n-2}]$. Thus, the product only uses about half of the total range $[-2^{2n-1} : 2^{2n-1} - 1]$ representable in two's complement with $2n$ bits, and overflows can only occur for very high values of $I(w)$.

From $I(p) \leq 2^{2n-2}$, we conclude that the only possibility for a positive overflow is for $I(w) \geq 2^{2n-2}$, and thus $I(w) \in \{2 \cdot 2^{2n-3}, 3 \cdot 2^{2n-3}\}$. The resulting error values are

$$\begin{aligned} e &\in \{2^{2n} - 2 \cdot 2^{2n-3}, 2^{2n} - 3 \cdot 2^{2n-3}\} \\ &= \{3 \cdot 2^{2n-2}, 5 \cdot 2^{2n-3}\}. \end{aligned}$$

Because of $I(p) > -2^{2n-2}$, a negative overflow can only occur for $I(w) < -2^{2n-2} - 1$, i.e. $I(w) \in \{-3 \cdot 2^{2n-3}, -2^{2n-1}\}$. The resulting error values are

$$\begin{aligned} e &\in \{2^{2n} - 3 \cdot 2^{2n-3}, 2^{2n} - 2^{2n-1}\} \\ &= \{5 \cdot 2^{2n-3}, 2^{2n-1}\} \end{aligned}$$

The following lemma summarizes the above calculations.

Lemma 3 *Let M be an n-bit multiplier constructed as given in Lemma 2. Then*

$$E(M) \subset \{\alpha 2^i | \alpha \in [1:3], i \in [0:2n-2]\} \cup \{5 \cdot 2^{2n-3}\}.$$

From Lemma 1 and Lemma 3 it follows that[7].

Theorem 1 *A check base b achieves fault secureness for an arbitrary multiplier constructed as given in Lemma 2 if:*

$$b \in \mathbf{N} \setminus \{\gamma 2^i | \gamma \in [1:5],\ i \in \mathbf{N}_0\}$$

Thus, the smallest check base to achieve fault secureness for arbitrary sized multipliers with respect to our model is $b = 7$.

Since in practice modulo 3 checking is very popular, let us reconsider our analysis for $b = 3$. Assume that *the FAs and HAs have been realized such that error values of the form ± 3 do not occur, i.e. a fault can not simultaneously flip both outputs in the same direction.* Applying the same error analysis as above it can be shown that under this 'restricted cellular fault model' for the set $E_{re}(M)$ of corresponding error values we have [33]:

$$E_{re}(M) \subset \{2^i | i \in [0:2n-1]\} \cup \{3 \cdot 2^{2n-2}\}.$$

It follows that there exists only one error value which can escape detection for modulo 3 checking: $3 \cdot 2^{2n-2}$. In addition, this error value can only occur for exactly one input combination $I(x) = I(y) = -2^{n-1}$ and only for faulty cell outputs on lines of significance 2^{2n-2} [33]. Thus, the occurrence of this error value is highly improbable and may be neglected for practical purposes. We conclude that modulo 3 checking is very efficient if the above assumption concerning the realization of the FAs and HAs is fulfilled.

[7]For the case that the result of a multiplication is not computed exactly but rounded with respect to an arithmetic base $a \in \mathbf{N}$, it has been shown in [30] that b must divide a. Note that this result does not apply here since the product is computed exactly in two's complement representation.

4 Registers and Booth recoding

In this section we extend the original residue checking scheme in order to also detect faults in the input and output registers and Booth recoding circuitry. In addition, we will argue that these extensions can be omitted if a small latency in error detection is acceptable.

4.1 Register checking

There are two different possibilities for embedding a residue code checked multiplier in a self-checking system: (1) The complete system computes with residue encoded operands. (2) Residue codes are only applied locally for checking the multiplier. For the second scheme faults in the input registers can be checked by the code which is applied to the surrounding circuitry. For the first scheme residue checking also has to guarantee the detection of all register faults.

Clearly, faults of the output register only result in absolute error values of the form 2^i, $i \in [0 : 2n-1]$, and thus, can be detected with the original checking scheme by any check base which is no power of two. For faults of the input registers we have already argued in Example 2 that in order to detect the corresponding errors in the product we must chose $b > 2^{n-1}$, which means that we would nearly have to duplicate the circuit. The solution to this problem is to check for input register faults locally, i.e. directly at the register outputs, and not at the outputs of the multiplier.

One possibility to do this local checking is by adding a residue checker at the outputs of each register. The hardware cost of these residue checkers is relatively high compared to the cost of the input registers, which are normally just simple latches. Thus, it is preferable to duplicate the input registers and check their output by means of a two-rail checker.[8]

4.2 Booth recoding

For the multiplication schemes of Section 2.1 n partial products had to be added. In order to reduce the number of partial products by one half, modified Booth recoding can be applied [39, 18]. Since modified Booth recoding reduces the area requirements of the overall circuit it is used in most of today's parallel multipliers [25, 15, 16, 17]. The savings in area have to be paid by a slightly more complicated multiplier design as explained next.

For simplicity of presentation let us assume that n is even, generalization to the case n odd is straight forward. The multiplier x is transformed from two's complement representation into a signed digit representation $[w_{n-2}, \ldots, w_2, w_0]_{SD}$ of length $\frac{n}{2}$ such that:

$$\begin{aligned} & w_{n-2} \cdot 2^{n-2} + \cdots + w_2 \cdot 2^2 + w_0 \cdot 2^0 = I(x) \\ & \text{and} \;\; w_j \in [-2:2] \end{aligned} \tag{3}$$

This is done by setting

$$w_j := -2x_{j+1} + x_j + x_{j-1} \tag{4}$$

for $j \in \{0, 2, \ldots, n-2\}$, where $x_{-1} := 0$. It can be easily seen that for these settings Equation 3 holds [18]. (As an example, the recoding of $x = 100111$ is $[-2, 2, -1]_{SD}$ and $I(x) = -2^5 + 2^2 + 2^1 + 2^0 = w_4 \cdot 2^4 + w_2 \cdot 2^2 + w_0 \cdot 2^0 = (-2 \cdot 2^4) + (2 \cdot 2^2) + (-1 \cdot 2^0)$.)

Due to this recoding, only $\frac{n}{2}$ partial products pp'_j, $j \in \{0, 2, \ldots, n-2\}$, have to be summed up in two's complement representation, where $I(pp'_j) = w_j \cdot 2^j \cdot I(y)$.

Multipliers with Booth recoding differ from the designs of Section 2.1 in two aspects: (1) A *recoder* is added which transforms the multiplier x from two's complement representation into the signed digit representation $[w_{n-2}, \ldots, w_2, w_0]_{SD}$. (2) The functionality of the PPG cells has to be enhanced, since negation is now possible for all partial products and multiplication by two has to be accomplished.

The addition of the partial products (PPR part and CPA) is done following the same principles as in Section 2.1 (see [18] for full sign extension and correction terms and [16] for CSA folding). Especially, all the results proven for residue code checking in these parts are also valid for Booth recoded multipliers. As a consequence, it suffices to consider the recoder and the PPG part in the following.

Recoder Let us first consider the realization of the recoder. Each w_j is usually encoded with three bits $w_{j,2}w_{j,1}w_{j,0}$ in signed binary notation, i.e. $w_j = (-1)^{w_{j,2}} \cdot (2w_{j,1} + w_{j,0})$ [39]. The corresponding bits are computed as:

$$\begin{aligned} w_{j,0} &= x_j \oplus x_{j-1} \\ w_{j,1} &= x_{j+1}\overline{x_j}\,\overline{x_{j-1}} \vee \overline{x_{j+1}}\,x_j x_{j-1} \\ w_{j,2} &= x_{j+1} \end{aligned} \tag{5}$$

Figure 10 shows an example of an eight bit recoder. Each *rec* cell computes one signed digit of the recoding. Since $x_{-1} := 0$, the design of the cell rec' computing the least significant digit is simplified.

[8] Alternatively we may also check the input registers by locally generating a parity bit. In this case we have to add a parity generator, one additional register cell, and a parity checker for each register.

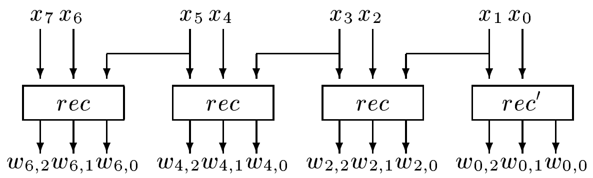

Figure 10: Eight bit Booth recoder

rec and rec' will be considered as basic cells in what follows, i.e. we allow for arbitrary combinational faults inside these cells. As an example for a recoder error assume that a fault causes $w_{0,2}w_{0,1}w_{0,0}$ to change from 100 ($w_0 = -0$) to 101 ($w_0 = -1$). Then instead of the recoding for $I(x)$ the recoding for $I(x)-1$ is computed, and we have the same effect as observed in Example 2 for the input register, i.e. $e = |I(x) \cdot I(y) - (I(x)-1) \cdot I(y)| = |I(y)| \in [0 : 2^{n-1}]$, and thus, $b > 2^{n-1}$ if we check for these faults at the product outputs.

This problem is solved by also duplicating the recoding logic[9], and deferring the two-rail check for the x-register to the outputs of the Booth recoder. The resulting configuration is depicted in Figure 11, where dark circles indicate that the corresponding outputs are inverted. (Note, that the three error signals of Figure 11 can be easily reduced to one error signal by means of a two-rail checker, which has been omitted for simplicity.)

It is also possible to use a single input register and Booth recoder and do residue checking at the outputs of the recoder. But residue computation for the signed digit representation of a number is even more costly than for its two's complement representation. Thus, the cost advantages for saving the register and the Booth recoder have to be paid again for the residue computation. In addition, for achieving the self-testing property (for modulo 7 checking) we need a more complicated wiring scheme than the one proposed in Section 5 for the two-rail checker. For brevity, we omit details here.

Partial product generation Consider now the *PPG part.* The multiplication by 2 necessary for $|w_j| = 2$ is realized by a shift operation of the PPG cells. As a result, neighboring PPG cells may share some logic as indicated by the PPG cell design shown in Figure 12, where the *exor*-gate for operand inversion is shared between neighboring cells. Nevertheless, since $w_{j,1}w_{j,0} = 11$, i.e. $|w_j| = 3$, can not occur for a fault free recoder (see Equation 4), a faulty *exor*-cell determines exactly one partial product bit for a specific input combination. Thus, we obtain the same result as in Section 3, i.e. only error values of the form 2^i can result on the partial products.

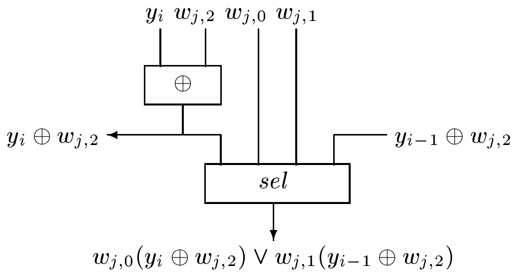

Figure 12: PPG cell design for Booth recoding

4.3 Cost reduction

Let us reconsider the scheme of Figure 11. A large fraction of the overhead for on-line error detection is due to the checks for the registers and recoder which constitute only a small portion of the overall circuit. The following lemma shows that, if we are willing to accept only a small latency for the detection of these faults, the above checks can be omitted.

Lemma 4 *Consider a (non redundant) cell fault f of the input registers or the Booth recoder. Let $p(f)$ denote the probability of detecting f by a* modulo 7 check at the outputs of the multiplier *under the assumption that all input combinations to the multiplier are equally probable, then:*

$$p(f) \geq \frac{1}{64}$$

Proof: Let us first consider a fault of one of the input registers: Wlog. assume that the $(i+1)$-th flip-flop of the x-register is faulty. The probability of locally exercising this fault is greater or equal to $\frac{1}{4}$, i.e. we need a specific pair of input and internal state of the register cell. Assume x and y are the correct operands. Since the fault flips bit x_i, the absolute error value of the product is $2^i \cdot I(y)$. Thus, the fault is detected by the modulo 7 check if and only if $(2^i \cdot I(y)) \bmod 7 \neq 0$. Since $2^i \bmod 7 \neq 0$, this is equivalent to $I(y) \bmod 7 \neq 0$, the probability of which is approximately $\frac{6}{7}$. As a consequence, $p(f) \approx \frac{1}{4} \cdot \frac{6}{7} \geq \frac{1}{64}$.

Consider now the recoder and assume that the cell computing w_i is faulty. Since this cell is combinational and has 3 inputs the probability of locally stimulating the fault is at least $\frac{1}{8}$. Let w_i^f denote the faulty cell

[9] Part of the duplication overhead for the Booth recoder can be saved by applying technology dependent two-rail design techniques at the transistor level as presented in [40, 4].

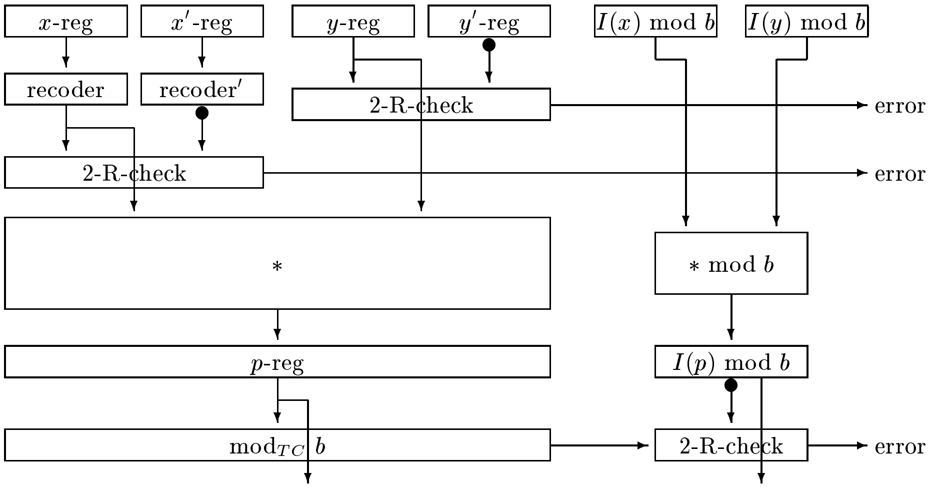

Figure 11: Checking scheme for Booth recoded multiplier

output and assume that $w_i^f \notin \{+3, -3\}$, i.e. $w_{i,1}^f w_{i,0}^f \neq 11$. Then $|I(p) - I(p^f)| = \alpha \cdot I(y) \cdot 2^i$, where $\alpha \in [1:4]$. Since 7 does not divide α and 2^i, we conclude as above that the erroneous output is detected with a probability of approximately $\frac{6}{7}$, and $p(f) \approx \frac{1}{8} \cdot \frac{6}{7} \geq \frac{1}{64}$.

The only case left is that of a faulty recoder cell which only produces erroneous outputs of the form $w_i^f \in \{+3, -3\}$. This case can not be handled as the other ones above due to the following fact: Since the value $w_i \in \{+3, -3\}$ does not occur during normal operation of the circuit, the logical operation of the PPG cells for this input can be chosen arbitrarily. Thus, in order to minimize hardware, the PPG cell given in Figure 12 does not compute a times 3 multiple for input $w_{i,1}w_{i,0} = 11$ but the logical 'or' of the times 2 and times 1 multiples. As a consequence, the discussion of this case becomes very technical. Since it reveals no essential new insights, we omit it here for brevity. ∎

The assumption of equally probable input combinations in the above lemma needs some further discussion. The proof reveals that we can replace this assumption by the following informal demands:
(1) For fault propagation it is necessary that none of the operands is 'restricted' to multiples of 7. (2) For fault activation we need that: (a) The value of each input line is neither 'restricted' to zero nor to one (register faults). (b) All input combinations are likely to occur for neighboring positions of the x-operand (the inputs to a *rec* or *rec'* cell of the Booth recoder).

The first demand will be fulfilled for nearly all applications. Also (2a) is not very restrictive, since it will normally be met by applications with varying signs of the operands. The only critical assumption is (2b), which implies frequent occurrence of operands with high absolute value. Thus depending on the application it might be preferable to only save the duplication of the registers and the two-rail checker for the y-register, but retain the duplicated recoder and its checker.

5 Check logic construction

Let C be a self-checking checker with input code space $\mathcal{I}$, output code space $\mathcal{O}$, and set of normal inputs N. The task of C is to signal any appearance of a noncode input by a noncode output.

Definition 3 *C is* code disjoint *for input code $\mathcal{I}$ and output code $\mathcal{O}$ if and only if:*

$$C(i) \in \mathcal{O} \Longleftrightarrow i \in \mathcal{I}$$

Besides being code disjoint, the checker also has to signal internal faults. This can be guaranteed by building C such that it is self-testing with respect to its set N of normal inputs (see Definition 2).

We start this section by shortly reviewing standard techniques for building the checking logic of Figure 11. Since the code disjoint property does not depend on the specific environment (given by the set N of normal inputs to the checker), it follows directly from standard proofs [27]. This is not true for the self-testing property which depends on N. How to structure the checkers in order to guarantee the self-testing property for their specific environment, will be discussed in the second part of this section.

5.1 Basic checking techniques

Two-rail checking Let $0^* := 01$ and $1^* := 10$. The set $TR_l := \{0^*, 1^*\}^l$ is called the *two-rail code* of length l. For a two-rail checker C we have $\mathcal{I} = TR_l$ and $\mathcal{O} = TR_1$, i.e. C gets as input l tuples $b_i^t := (b_i, b_i') \in \mathbf{B}^2$, $i \in [1:l]$, and reduces them to one output tuple $r^t := (r, r')$ such that:

$$\begin{aligned} r^t \in TR_1 &\iff (b_{l-1}^t, \ldots, b_0^t) \in TR_l \\ &\iff b_{l-1}' \ldots b_0' = \overline{b_{l-1}} \ldots \overline{b_0} \end{aligned}$$

The standard realization of a two-rail checker is as a tree of basic cells *trc* reducing two bit tuples to one [27]. The design of this cell is depicted in Figure 13. It can be easily verified that cell *trc* computes an output from TR_1 if and only if it receives an input from TR_2.

An example of a 12-input two-rail checker is given in Figure 14. Here, each circle corresponds to a *trc*-cell, and each line to a bus of width two. (The reason for shading two of the *trc*-cells will be explained later.)

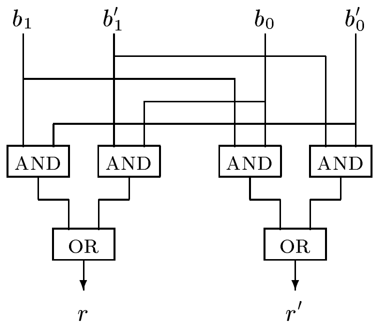

Figure 13: Realization of cell *trc*

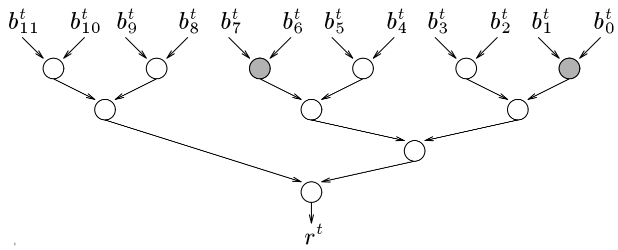

Figure 14: Realization of 12 input two-rail checker

Residue checking There are a lot of different possibilities for designing residue checkers depending on the check base b and speed/area requirements (see for example [34, 26, 41, 42, 43]). In the following, we will assume that $b = 7$, and the residue checker is built as shown in Figure 9 with the $\text{mod}_{TC}\, b$ circuit constructed as suggested in [29, 34, 26].

Let $m := 2n$ denote the bit width of the product. The $\text{mod}_{TC}\, 7$ circuit has to compute the residue of the product's two's complement interpretation, i.e.:

$$I(p) \bmod 7 = (-p_{m-1}2^{m-1} + \sum_{i=0}^{m-2} p_i 2^i) \bmod 7$$

Since 7 is a low cost check base, computation of $(\sum_{i=0}^{m-2} p_i 2^i) \bmod 7$ can be done by dividing $p_{m-2} \ldots p_0$ into $\lceil \frac{m-1}{3} \rceil$ groups of 3 bits starting from the right (the leftmost group may be of size less than 3), and adding these up with modulo 7 adders [29]. For the sign bit p_{m-1} it can be shown that

$$(-p_{m-1}2^{m-1}) \bmod 7 \quad = \quad p_{m-1} \cdot v$$

where $v = 6$ ($v = 5$) ($v = 3$) if $(m-1) \bmod 3 = 0$ ($(m-1) \bmod 3 = 1$) ($(m-1) \bmod 3 = 2$). For the proof of this fact we restrict to the case where $(m-1) \bmod 3 = 0$, the other cases are handled analogously. Let $m - 1 = 3\alpha$, $\alpha \in \mathbf{N}$, then

$$\begin{aligned} (-p_{m-1}2^{m-1}) \bmod 7 &= (-p_{m-1}(2^3)^\alpha) \bmod 7 \\ &= -p_{m-1} \bmod 7 \end{aligned}$$

since $2^3 \bmod 7 = 1$. From the fact that $-1 \bmod 7 = 6$ it then follows that $(-p_{m-1}2^{m-1}) \bmod 7 = p_{m-1} \cdot 6$.

Figure 15 shows an example of a residue computation tree for $m = 14$. The modulo 7 adders (denoted by $+$ in Figure 15) are normally designed as 3-bit adders with end-around carry. (Note, that care should be taken not to create a feed-back loop by the end-around carry in order to avoid sequential and indeterminate behavior [44].) The leftmost adder denoted by $+'$ is a simplified modulo 7 adder which computes $p_{13} \cdot 5 + p_{12}$. (Since $13 \bmod 3 = 1$, it follows that $(-p_{13}2^{13}) \bmod 7 = p_{13} \cdot 5$.) As an example, for input vector $p = 101\ldots1$ the residue computation tree of Figure 15 computes:

$$\begin{aligned} &I(101\ldots1) \bmod 7 = (-2^{13} + \sum_{i=0}^{11} 2^i) \bmod 7 \\ &= (5 + 0 + 7 + 7 + 7 + 7) \bmod 7 = 5 \end{aligned}$$

For a modulo 7 adder realized as a 3-bit adder with end-around carry there are two representations of zero, namely $0 = 000$ and $7 = 111$. In order to have a singular representation for comparison purposes, we add a circuit at the outputs of the residue tree which maps 111 to 000.

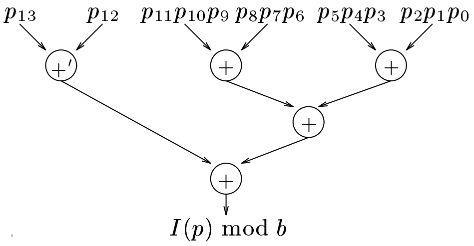

Figure 15: Residue computation for 14-bit product

5.2 Self-testing property

Two-rail checkers It is well known [27] that for the realization of a two-rail checker C as given in Figures 13 and 14 it holds that:

Lemma 5 *C is self-testing (with respect to stuck-at faults*[10]*) for a set of normal inputs $N \subset \{0^*, 1^*\}^l$ $\Longleftrightarrow$ N applies all input combinations from $\{0^*, 1^*\}^2$ to every trc-cell of C.*

Clearly, the above property is fulfilled for the two-rail checker monitoring the outputs of the duplicated y-register (see Figure 11), since all values from $\{0^*, 1^*\}^n$ can occur at the outputs of this register during normal operation.

Consider now the two-rail checker at the outputs of the duplicated Booth recoder in Figure 11. By Equations 4 and 5 we can determine the set of normal outputs for the basic cells of a recoder (see Figure 10).

(1) *rec'*-cell: Since $x_{-1} := 0$ there are only four possible input combinations $x_1x_0x_{-1} = 000, 010, 100, 110$. The corresponding outputs are $w_0 = +0, +1, -2, -1$, i.e.:

$$w_{0,2}w_{0,1}w_{0,0} = 000, 001, 110, 101 \qquad (6)$$

(2) *rec*-cell: Since x_{j-1} can also be set to one, w_j can assume all values from $\{-2, -1, -0, +0, +1, +2\}$. Thus, the possible output combinations are:

$$w_{j,2}w_{j,1}w_{j,0} = 000, 001, 110, 101 \text{ and } 010, 100 \qquad (7)$$

From the above analysis we obtain that at the output pairs $w_{0,2}w_{0,1}$, and $w_{j,1}w_{j,0}$, $j \geq 0$, not all values from $\mathbf{B}^2$ are possible. Thus, if we apply the two-rail checker of Figure 14 for checking the duplicated Booth recoder of Figure 10, the shaded cells would not receive all possible input combinations from $\{0^*, 1^*\}^2$, and by Lemma 5 the two-rail checker is not self-testing.

[10]Since the self-testing property with respect to cellular faults can not be achieved for the two-rail checker we restrict to stuck-at faults in the following.

This problem can be solved by observing that the output tuples of the duplicated Booth recoder can be fed into the two-rail checker in an arbitrary order. By permuting them appropriately the self-testing property can be guaranteed. Figure 16 gives a regular permutation which achieves this goal. Here, C denotes an arbitrary planar tree of *trc*-cells. The leftmost input configuration only occurs if the recoder consists of an odd number of cells. A self-testing scheme for the 12-output Booth recoder of Figure 10 is given in Figure 17.

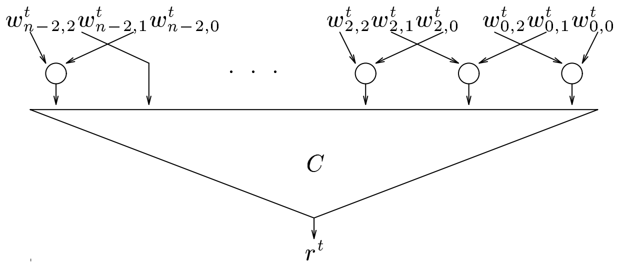

Figure 16: Self-testing two-rail checker

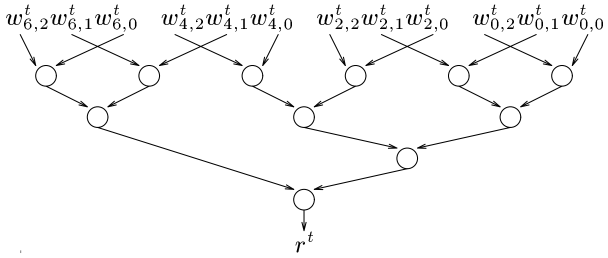

Figure 17: Self-testing example for 12 inputs

Theorem 2 *A two-rail checker for the duplicated Booth recoder (with at least two recoder cells) is self-testing if it is structured according to the scheme given in Figure 16.*

Proof: (Sketch)
Let us first consider the *trc*-cells of the first level in Figure 16 which do not belong to C:
From the output behavior of the *rec'*- and *rec*-cells (see Equations 6 and 7) it follows immediately that all input combinations from $\{0^*, 1^*\}^2$ are applicable to the cells connected to inputs $w^t_{j,2}$ and $w^t_{j,0}$, $j \in \{0, 2, 4, \ldots\}$.
Now consider the *trc*-cells of the first level connected to inputs $w^t_{j+2,1}$ and $w^t_{j,1}$, $j \in \{0, 4, 8, \ldots\}$. Clearly,

any value from $\{0^*, 1^*\}$ can be applied to the right input $w^t_{j,1}$ of such a cell by choosing $x_{j+1}x_jx_{j-1}$ appropriately (see Equations 6 and 7). The value of $w^t_{j+2,1}$ depends on inputs $x_{j+3}x_{j+2}x_{j+1}$. Thus, when fixing $x_{j+1}x_jx_{j-1}$, we are still free to choose the values of x_{j+3} and x_{j+2}, and by doing so any value can be applied to $w^t_{j+2,1}$ for given x_{j+1} (see Equation 5).

For brevity we omit the proof for the *trc*-cells of C. It is based on the fact that *trc*-cells compute the 'exor-function' on code word inputs [27], and can be found in [33]. ■

Residue checker First consider the two-rail checker which compares the residue computed from the final product to the inverted result of the multiplier modulo b (see Figure 9). For this checker the set N of normal inputs is equal to $\{0^*, 1^*\}^3 \setminus \{1^*1^*1^*\}$. Thus, it can be easily seen that N fulfills Lemma 5 and is self-testing. (This is not true for modulo 3 checking. Techniques for achieving the self-testing property in this case can be found in [34, 45].)

Let us now look at the residue computation tree. In order to be independent of implementation details for the modulo 7 adders, our aim is to achieve the self-testing property with respect to the cellular fault model, considering the modulo 7 adders and the circuit mapping 111 to 000 as basic cells.

Since the product p can only assume values from the subset $[-2^{n-1} \cdot (2^{n-1} - 1) : 2^{2n-2}]$, not all input combinations are applicable to the residue tree during normal operation (see Example 3). The set of possible input combinations can be exactly characterized as follows:
The number representations corresponding to $I(p) \in [0 : 2^{2n-2}]$ are given by:

$$P_p := \{010\ldots0\} \cup \{00w | w \in \mathbf{B}^{m-2}\}$$

For the possible negative product values $I(p) \in [-(2^{2n-2} - 2^{n-1}) : -1]$ the corresponding number representations are:

$$P_n := \{11w | (w \in \mathbf{B}^{m-2}) \wedge (\exists i \geq n-1 : w_i = 1)\}$$

Thus, if we structure the residue tree as given in Figure 15, then the input combination $(1, 0)$ would not be applicable to the leftmost $+'$ adder, and the self-testing property with respect to the cellular fault model is not achieved.

Again, as in the case of the two-rail checker for the Booth recoder, this problem can be solved by permuting the inputs to the residue tree appropriately. Since there are no input restrictions for the least significant bit positions (see set P_p), it is sufficient to only permute the leading bits. A schematic diagram of the corresponding scheme is given in Figure 18. Here, R denotes an arbitrarily structured tree of modulo 7 adders with added $111 \rightarrow 000$ mapping circuit. $+'$ and $+''$ are appropriately specialized versions of the modulo 7 adder.

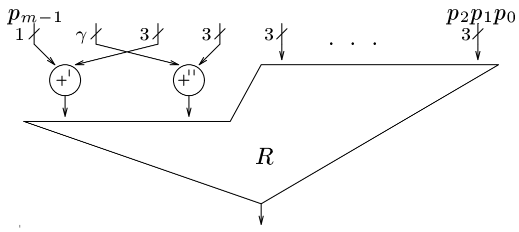

Figure 18: Schematic for structuring the residue tree

Lemma 6 *Let $n > 4$. If the adder tree for computing the modulo 7 residue of the product is structured as given in Figure 18, then during normal operation all possible input combinations are applicable to its basic cells.*

Proof: (Sketch)
Consider the $+'$-cell. Clearly, all values of the form $(0, v)$, $v \in \mathbf{B}^3$, are applied to the inputs of this cell by the number representations from P_p. If $n \geq 5$, i.e. $m \geq 10$, then the representations from P_n guarantee that all input combinations $(1, v)$, $v \in \mathbf{B}^3$, are applied to $+'$. A similar argumentation can be done for cell $+''$.

Consider now the cells of R. Obviously any input combination is possible for the $000 \rightarrow 111$ mapping cell. For an arbitrary $+$-cell z in R let T_l (T_r) denote the adder tree computing its left (right) input. Then we can apply $(u, v) \in \mathbf{B}^6$ to z by setting the rightmost input of T_l (T_r) to u (v) and all other inputs to zero. Obviously, such an input combination exists in P_p. ■

In order to prove that the residue computation tree of Figure 18 is self-testing, we also have to show that all faulty cell responses are propagated. This is clearly true for differences v/v^f, $v \neq v^f$, such that $v/v^f \notin \{000/111, 111/000\}$. The differences 000/111 and 111/000 can not be propagated since they are masked by the $111 \rightarrow 000$ mapping circuit. Thus, the self-testing property can only be achieved for cell faults which exhibit at least one faulty behavior different from 000/111 and 111/000.

Now consider a fault which only causes a cell to output 111 instead of 000 or vice versa. Clearly, such a fault doesn't corrupt the trees ability for correct residue computation. As a consequence, the residue checker performs its desired function of indicating the occurrence of any noncode input even in the presence of faults of this type.

To allow for situations as the one described above, the notion of *strongly code disjoint* checkers has been introduced in [46].

Definition 4 *Let C be a self-checking checker with input code space $\mathcal{I}$, output code space $\mathcal{O}$, and set of normal inputs N. The fault set of C is denoted by F.*

C *is* strongly code disjoint $\Longleftrightarrow$

(a) C is code disjoint with respect to codes $\mathcal{I}$ and $\mathcal{O}$.

(b) If $<f_1, \ldots, f_k>$ is a multiple fault from F such that for every $i \in \{1, \ldots, k\}$ the circuit $C^{<f_1,\ldots,f_i>}$ is not self-testing for input set N, then:

$C^{<f_1,\ldots,f_k>}$ maps any noncode input to a noncode output.

For a formal proof that strongly code disjoint checkers provide a sufficient level of security see [46]. From the above definition it immediately follows:

Theorem 3 *Let C be a residue checker for the final product as given in Figure 9.*

If the residue computation tree is built according to the scheme given in Figure 18, then C is strongly code disjoint.

6 Overhead estimation

We have analyzed several examples of residue code checked multipliers in order to determine the hardware overhead for checking and its dependence on bit width. The hardware overhead is measured as $\frac{\#cl}{\#om} \cdot 100\%$, where $\#om$ ($\#cl$) denotes the number of gates in the original multiplier (the additional check logic) including the register latches. In order for this analysis to be valid for all the schemes given in Section 2.1 (i.e. full sign extension, addition of correction terms, CSA folding), and arbitrary structure of the carry propagate adder, the number of gates in the original multiplier $\#om$ has been approximated by a lower bound which is valid for all of these architectures. Thus the results computed are an upper bound on the actual gate overhead for checking. The results for a specific architecture can be slightly better.

Table 1 summarizes the results obtained. The first column specifies the bit width of the multiplier. Part 1 (2) of the table gives the results for multipliers without (with) Booth recoding. For each of these parts the lat.- (no lat.-) column gives the hardware overhead for the case that a latency of 64 operations in error detection is (is not) allowed. All overheads are with respect to modulo 7 checking.

	1.) no Booth rec.		2.) Booth rec.	
width	lat.	no lat.	lat.	no lat.
16	18.4 %	31.1 %	22.2 %	42.8 %
32	7.8 %	14.1 %	9.9 %	20.7 %
64	3.5 %	6.6 %	4.5 %	10 %
128	1.7 %	3.2 %	2.2 %	5 %

Table 1: Hardware overheads

As can be observed from Table 1, the hardware overhead reduces by approximately one half when latency is allowed, and the duplication checks for the input registers and the recoder are omitted. Since for Booth recoded multiplication a larger amount of checking hardware can be omitted, the savings are slightly higher in this case. Comparing part 1 and 2 we notice that the overhead for Booth recoded multipliers is in general higher. There are two reasons for this effect: First, the gate count $\#om$ of the original multiplier is less for recoded multiplication. Second, the duplication checks are more costly (applies only for the 'no latency' case). For all the configurations the hardware overhead is approximately divided by two as the bit width is doubled. This is due to the fact that the size of the original multiplier $\#om$ grows proportional to n^2 while the check logic size $\#cl$ only increases linearly with n.

Up to now, we have assumed that the input operands of the multiplier are already encoded. Since memory systems are usually checked with parity codes, there has to be some residue code generation circuitry for computing this encoding. The exact overhead induced by the encoding circuitry depends on the specific architecture of the overall system in which the multiplier is embedded. If the complete system computes with residue encoded operands, conversion from parity code to residue code has only to be done once, i.e. the residue generation circuitry and its overhead can be shared between all modules. Also, depending on whether the operands are read from memory sequentially or in parallel, one or two residue code generators may be necessary. In an extreme case, the complete system computes with a parity code and only the multiplier is checked by residue codes. In this situation the residue generation circuits could not be shared with other modules. But note that for this case the duplication of input registers can be saved, thus reducing the overhead numbers of Table 1 for the no latency case. This is due to the fact that the registers can be checked by the parity code which is available anyway.

Table 2 gives an upper bound on the additional hardware overhead which can result from operand encoding, i.e. we assume the worst case where two residue code generation circuits are needed and their overhead can not be shared with other modules. The observations made for Table 1 also apply here, i.e. the encoding overhead is slightly higher for multipliers with Booth recoding, and it is approximately divided by two as the operand length is doubled.

width	no Booth rec.	Booth rec.
16	12.3 %	14.8 %
32	6.6 %	8.3 %
64	3.1 %	4.0 %
128	1.6 %	2.1 %

Table 2: Additional overhead for operand encoding

7 Conclusion

In this paper we have studied the efficiency of residue codes for checking parallel two's complement multipliers. In contrast to previous work [10, 6, 7] our approach is not simulation based but analytical, i.e. we exactly characterize the dependency between internal faults and the erroneous output values which can result from them.

Three different multiplication schemes have been considered: full sign extension, addition of correction terms and CSA folding. For all of them 7 has been proven to be the smallest check base which achieves fault secureness. But we have also argued, that check base 3 is very efficient if full and half adders are realized appropriately. For the input registers and the Booth recoding circuitry it has been shown that the original checking scheme has to be enhanced by additional checks to guarantee fault secureness. These checks can be omitted if a small latency in error detection (less than 64 operations on the average) is admitted. In order for a circuit to be totally self-checking, also the self-testing property has to be guaranteed. We have shown that this property is not fulfilled for standard checking logic, but can be achieved by restructuring the checkers appropriately.

Evaluation of the hardware overhead shows that residue checking offers a very cost effective way for on-line error detection in two's complement multipliers. This is especially true for large operand sizes, since the hardware overhead is approximately divided by two when the bit width is doubled.

References

[1] Y. Thoma. Coding techniques in fault-tolerant, self-checking, and fail-save circuits. In D.K. Pradhan, editor, *Fault-Tolerant Computing: Theory and Techniques*, vol. 1, pages 336–416. Prentice-Hall, 1986.

[2] S. Manich, C. Ruiz, and J. Figueras. A fault-secure modified booth multiplier. In *Proc. of the EDAC-EUROASIC 93*, pages 180–183, 1993.

[3] N. Takagi and S. Yajima. On-line error-detectable high-speed multiplier using redundant binary representation and three-rail logic. *IEEE Trans. on Comp.*, C-36:1310–1317, November 1987.

[4] M. Nicolaidis and H. Bederr. Efficient implementations of self-checking multiply and divide arrays. In *Proc. of the European Design and Test Conf.*, pages 574–579, 1994.

[5] J.-C. Lo, S. Thanawastien, and T.R.N. Rao. Concurrent error detection in arithmetic and logical operations using berger codes. In *Proc. of the 9th Symp. on Computer Arithmetic*, pages 233–240, 1989.

[6] I.L. Sayers and D.J. Kinniment. Low-cost residue codes and their application to self-checking VLSI systems. *IEE Proc. E*, 132(4):197–202, 1985.

[7] W.H. Debany, A.R. Macera, D.E. Daskiewich, M.J. Gorniak, K.A. Kwiat, and H.B. Dussault. Effective concurrent test for a parallel-input multiplier using modulo 3. In *Proc. of the 1992 VLSI Test Symp.*, pages 280–284, 1992.

[8] J.H. Patel and L.Y. Fung. Concurrent error detection in multiply and divide arrays. *IEEE Trans. on Comp.*, C-32:417–422, April 1983.

[9] Y.-M. Hsu and E.E. Swartzlander. Time redundant error correcting adders and multipliers. In *Proc. of the IEEE Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 247–256, 1992.

[10] R.W. Heckelman and D.K. Bhavsar. Self-testing VLSI. In *Proc. of the 1981 IEEE Int. Solid-State Circuits Conf.*, pages 174–175, 1981.

[11] T.J. Slegel and R.J. Veracca. Design and performance of the IBM Enterprise System/9000 Type 9121 vector facility. *IBM J. Res. Develop.*, 35:367–381, 1991.

[12] J.W. Watterson and J.J. Hallenbeck. Modulo 3 residue checker: New results on performance and cost. *IEEE Trans. on Comp.*, C-37:608–612, May 1988.

[13] T.J. Brosnan and N.R. Strader. Modular error detection for bit-serial multiplication. *IEEE Trans. on Comp.*, C-37:1043–1052, September 1988.

[14] C.R. Baugh and B.A. Wooley. A two's complement parallel array multiplication algorithm. *IEEE Trans. on Comp.*, C-22:1045–1047, 1973.

[15] K.F. Pang, H.W. Soong, R. Sexton, and P.H. Ang. Generation of high speed CMOS multiplier-accumulators. In *Proc. of the* 1988 *IEEE Int. Conf. on Computer Design*, pages 217–220, 1988.

[16] R. Stans. The testability of a modified booth multiplier. In *Proc. of the 1989 European Test Conf.*, pages 286–293, 1989.

[17] J. Fadavi-Ardekani. m×n booth encoded multiplier generator using optimized wallace trees. In *Proc. of the 1992 IEEE Int. Conf. on Computer Design*, pages 114–117, 1992.

[18] I. Koren. *Computer Arithmetic Algorithms*. Prentice-Hall, 1993.

[19] F.F. Sellers, M.Y. Hsiao, and L.W. Bearnson. *Error Detecting Logic for Digital Computers*. McGraw-Hill, 1968.

[20] G.G. Langdon and C.K. Tang. Concurrent error detection for group look-ahead binary adders. *IBM J. Res. Develop.*, 14(5):563–573, 1970.

[21] U. Sparmann. On-line error detection in fast adders by residue codes. Technical report TR 13/1992, Sonderforschungsbereich 124, Fachbereich Informatik, Universität des Saarlandes, 66041 Saarbrücken, FRG, 1992.

[22] U. Sparmann. On the check base selection problem for fast adders. In *Proc. of the 1993 VLSI Test Symp.*, pages 62–65, 1993.

[23] C.S. Wallace. A suggestion for a fast multiplier. *IEEE Trans. on Electronic Comp.*, EC-13:14–17, 1964.

[24] W.K. Luk and J. Vuillemin. Recursive implementation of optimal time VLSI integer multipliers. In *Proc. of the IFIP Congress 83*, pages 155–168, Amsterdam, 1983.

[25] D. Zuras and W.H. McAllister. Balanced delay trees and combinatorial devision in VLSI. *IEEE Journal of Solid-State Circuits*, SC-21:814–819, October 1986.

[26] J. Wakerly. *Error Detecting Codes, Self-Checking Circuits and Applications*. North Holland, 1978.

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

[28] M. Gössel and S. Graf. *Error Detection Circuits*. McGraw-Hill, 1993.

[29] A. Avizienis. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. *IEEE Transactions on Comp.*, C-20:1322–1331, 1971.

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

[31] G. Russell and I.L. Sayers. *Advanced Simulation and Test Methodologies for VLSI Design*. Van Nostrand Reinhold Int., 1989.

[32] N. Takagi and S. Yajima. An on-line error-detectable array divider with a redundant binary representation and a residue code. In *Proc. of the 18th Int. Symp. on Fault-Tolerant Comp.*, pages 174–179, 1988.

[33] U. Sparmann and S.M. Reddy. On the effectiveness of residue code checking for parallel two's complement multipliers. Technical Report 8-21-93, Electrical and Computer Engineering Department, University of Iowa, Iowa City, Iowa 52242, USA, 1993.

[34] M.J. Ashjaee and S.M. Reddy. On totally self-checking checkers for seperable codes. *IEEE Trans. on Comp.*, C-26:737–744, 1977.

[35] O.J. Bedrij. Carry-Select Adder. *IRE Trans. on Electronic Comp.*, pages 340–346, 1962.

[36] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the Association for Comp. Machinery*, 27(4):831–838, October 1980.

[37] B.W.Y. Wei and C.D. Thompson. Area-time optimal adder design. *IEEE Trans. on Comp.*, 39:666–675, 1990.

[38] J. Sklansky. Conditional-sum addition logic. *IRE-EC*, 9:226–231, 1960.

[39] O. Spaniol. *Computer arithmetic: logic and design*. Wiley, 1981.

[40] M. Nicolaidis. Efficient implementation of self-checking adders and ALUs. In *Proc. of the 23nd Int. Symp. on Fault-Tolerant Computing*, pages 586–595, 1993.

[41] D. Nikolos, A.M. Paschalis, and G. Philokyprou. Efficient design of totally self-checking checkers for all low-cost arithmetic codes. *IEEE Trans. on Comp.*, C-37:807–814, July 1988.

[42] S.J. Piestrak. Design of high-speed and cost-effective self-testing checkers for low-cost arithmetic codes. *IEEE Trans. on Comp.*, C-39:360–374, March 1990.

[43] S.J. Piestrak. Self-testing checkers for arithmetic codes with any check base A. In *Proc. of the Pacific Rim Int. Symp. on Fault Tolerant Systems*, pages 162–167, 1991.

[44] J.J. Shedletsky. Comment on the sequential and indeterminate behavior of an end-around-carry adder. *IEEE Trans. on Comp.*, pages 271–272, 1977.

[45] S. Kundu and S.M. Reddy. Embedded totally self-checking checkers: A practical design. *IEEE Design and Test of Comp.*, pages 5–12, August 1990.

[46] M. Nicolaidis and B. Courtois. Strongly code disjoint checkers. *IEEE Trans. on Comp.*, C37:751–756, 1988.