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

U. Sparmann*
Computer Science Department
University of Saarland
66041 Saarbrücken, Germany

S. M. Reddy
Dep. of Elec. and Comp. Eng.
University of Iowa
Iowa City, IA 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.

1 Introduction

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 have been studied considerably in the literature: parity prediction in [Tho86, MRF93], two and three rail encoding in [TY87, Nic94], Berger codes in [LTR89] and residue codes in [DMD+92]. 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 method which achieves an overhead proportional to n, and thus becomes more and more attractive with increasing operand size compared to the above schemes, is residue code checking.[1] First industrial applications of residue codes for checking parallel multipliers have been reported in [HB81, SV91].

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 [WH88, BI88]. For parallel multiplication only simulation based analysis for specific architectures has been reported [HB81, DMD+92], which indicates that a small percentage of errors goes undetected. (As an example, in [DMD+92] 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 [BW73, PSSA88, Sta89, FA92, Kor93]. 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, 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.

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

[1] Note that, if time redundancy is admissible, overhead $O(n)$ can also be achieved by time redundant techniques like recomputing with shifted operands [PF83], or recomputing with duplication with comparison [HS92].

The paper is organized as follows. Section 2 gives an overview of parallel multiplier architectures, and introduces some basic definitions concerning self-checking circuits. Fault effects are studied in Section 3, where it is also shown that 7 is the smallest check base to achieve complete error detection. Section 4 considers faults in the input registers and Booth recoding circuitry. 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.

2 Preliminaries

This section explains the basic algorithmic principles of todays parallel two's complement multipliers, and reviews some definitions concerning self-checking circuits.

2.1 VLSI implementations of parallel multiplication

In the following we identify an α-bit binary string $z = z_{\alpha-1} \dots z_0$ 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 $x = x_{n-1} \dots x_0$ (the multiplier) and $y = y_{n-1} \dots y_0$ (the multiplicand) and computes the product $p = p_{2n-1} \dots p_0$ 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 in [Spa92, Spa93].

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 [PSSA88, FA92]. 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 [Kor93]. There is a great variety of possibilities for structuring such a CSA tree [Kor93]. The simplest scheme, well suited for pipelining, is the linear tree. For fast multiplication Wallace trees, or related more regular structures [LV83, ZM86] 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 [Kor93] for PPG to Section 4.

Full sign extension The partial products $pp_0, \dots, pp_{n-1}$ are generated in two's complement representation, i.e. $I(pp_0) = x_0 2^0 \cdot I(y), \dots, I(pp_{n-2}) = x_{n-2}2^{n-2} \cdot I(y)$, $I(pp_{n-1}) = -x_{n-1}2^{n-1} \cdot I(y)$. They are obtained by the following rules [Kor93]:
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]

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

For pp_{n-1} we additionally have to negate 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.

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

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 $[10101] \cdot [01101]$

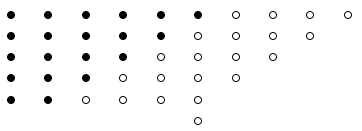

Figure 2: PPM with sign extensions

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.

There are many approaches to eliminate the sign extensions in the upper left triangle of the PPM, thus saving the

[2] z^α is short for $z \dots z$ α-times, and $x_i y_j := x_i \wedge y_j$.

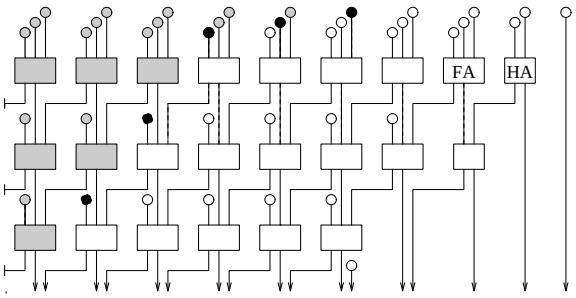

Figure 3: PPG and PPR with full sign extension

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 [BW73]. A slightly simpler method is given in [Kor93]. Since the scheme of [Kor93] is better suited for VLSI implementation and generalizes naturally to Booth recoded multipliers [Kor93], 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 [Kor93] 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 $[10101] \cdot [01101]$

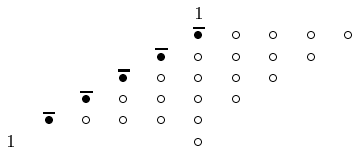

Figure 5: PPM with correction terms

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 [Sta89] in which all the shaded cells of Figure 3 have been eliminated.

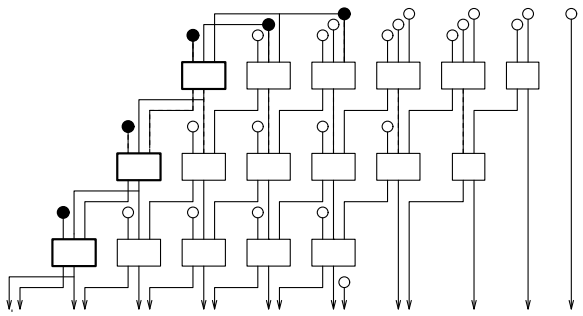

Figure 6: PPG and PPR with folded CSAs

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 shortly reviews some notions from the theory of self-checking circuits and residue codes which will be of importance in this paper. For further reference, see for example [SHB68, Wak78, RF89, GG93] for self-checking circuits, and in addition [Avi71, Rao74] for residue codes. Results on the effectiveness of residue codes for array dividers can be found in [TY88].

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 $I(x)$ mod b, $I(y)$ mod b and $(I(x) \cdot I(y))$ mod 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:

$$(p^f, I(p) \bmod b) \notin \mathcal{O}$$
$$\iff |I(p) - I(p^f)| \bmod b \neq 0. \qquad (1)$$

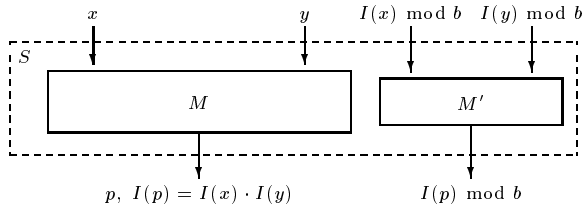

Figure 7: Multiplier checked by residue code

Note that two's complement adders are normally checked by a code of the form $(z, z \bmod b)$. Thus, a code transformation has to be done 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 $z = I(z) + z_{\alpha-1}2^{\alpha}$. For an example illustrating the importance of checking two's complement multiplication with code words of the form $(p, I(p) \bmod b)$ instead of $(p, p \bmod b)$ see [SR93].

For the **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.

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

Clearly, any erroneous output of M' results in a noncode word. Based on Equation 1, we thus obtain [Wak78]:

Lemma 1 *S is fault secure for F* $\iff$
$\forall e \in E(M) : e \bmod b \neq 0.$

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 compute the set $E(M)$ of absolute error values, for the multiplication schemes introduced in Section 2.1 (see Section 3). As a result of the analysis we will obtain that for $b = 7$ all error values 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.

Example 1 *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 [Wak78, RF89].

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[3], 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.

Example 2 *A block diagram of the residue checker monitoring the multiplier outputs is given in Figure 8 [AR77]. Here, $\bmod_{TC}$ b denotes a circuit computing the residue of its input in two's complement representation, i.e. $I(p)$ mod 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.*

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.

[3] 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.

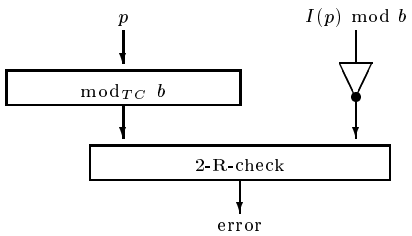

Figure 8: Residue checker

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. 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 [Span81] 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 [PSSA88, FA92]. In [Spa92] based on [WT90] a family of adders (ADD_{DP}) has been characterized which is very powerful (including carry select and conditional sum [Kor93] 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 [Spa92].*

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

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

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

$$e \in \{2^{2n} - 3 \cdot 2^{2n-3}, 2^{2n} - 2^{2n-1}\} = \{5 \cdot 2^{2n-3}, 2^{2n-1}\}.$$

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:

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 [SR93]:

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

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} [SR93]. 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.

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 1 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.

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 [Span81, Kor93]. Since modified Booth recoding saves area for the realization of the PPR part at the cost of low expenses in the PPG part it is used in most of today's parallel multipliers [ZM86, PSSA88, Sta89, FA92].

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 $[w_{n-2}, \ldots, w_2, w_0]_{SD}$ such that:

$$\sum_{j=0\,(2)}^{n-2} w_j \cdot 2^j = I(x) \quad \text{and} \quad w_j \in [-2:2] \tag{3}$$

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 [Kor93]. (As an example, the recoding of $x = 100111$ is $[-2, 2, -1]_{SD}$ and $I(x) = -2^5 + 2^2 + 2^1 + 2^0 = \sum_{j=0\,(2)}^{4} w_j \cdot 2^j = (-2 \cdot 2^4) + (2 \cdot 2^2) + (-1 \cdot 2^0)$.)

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 [Kor93] for full sign extension and correction terms and [Sta89] 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. $w_j = (-1)^{w_{j,2}} \cdot (2w_{j,1} + w_{j,0})$ [Span81]. The corresponding bits are computed as:

$$w_{j,0} = x_j \oplus x_{j-1}, \quad w_{j,2} = x_{j+1}, \quad \text{and}$$
$$w_{j,1} = x_{j+1}\overline{x_j}\,\overline{x_{j-1}} \vee \overline{x_{j+1}}\,x_j x_{j-1} \qquad (5)$$

Figure 9 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.

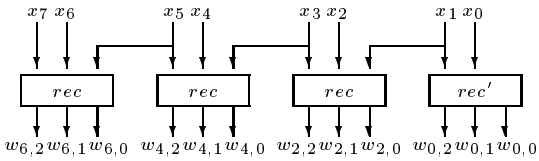

Figure 9: Eight bit Booth recoder

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 1 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[4], and deferring the two-rail check for the x-register to the outputs of the Booth recoder. The resulting configuration is depicted in Figure 10, where dark circles indicate that the corresponding outputs are inverted. (Note, that the three error signals of Figure 10 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 x 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.

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 11, 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.

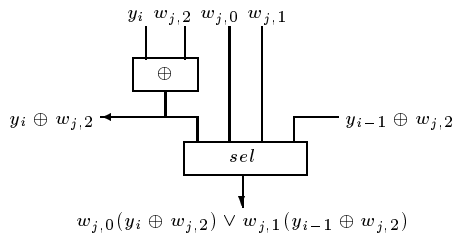

Figure 11: PPG cell design for Booth recoding

4.3 Cost reduction

Let us reconsider the scheme of Figure 10. 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

[4]Note, that 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 given in [Nic93, Nic94].

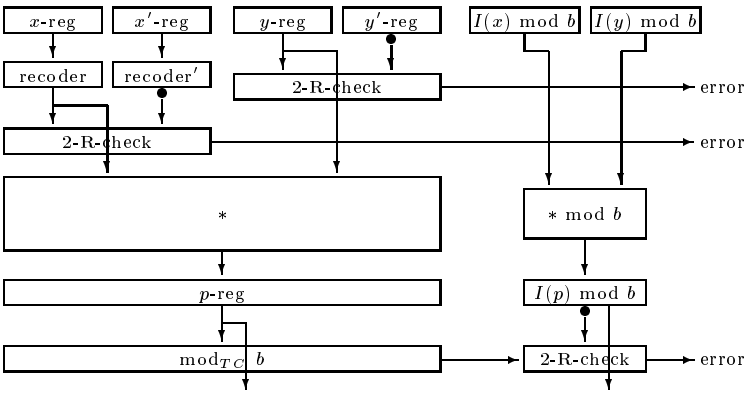

Figure 10: Checking scheme for Booth recoded multiplier

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 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, $p(f) \approx \frac{1}{4} \cdot \frac{6}{7} \geq \frac{1}{64}$.

The proof for the recoder cells can be found in [SR93]. We omit it here since it essentially follows the same argumentations as above. ■

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 cells). (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. In order to detect any appearance of a noncode input the checker must be code disjoint [RF89], i.e. $C(i) \in \mathcal{O} \Leftrightarrow i \in \mathcal{I}$. In addition, C 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). 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 [RF89]. This is not true for the self-testing property which depends on N. How to structure the checkers of Figure 10 in order to guarantee the self-testing property for their specific environment, will be discussed in the following.

Two-rail checkers 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 tupels $b_i^t := (b_i, b_i') \in \mathbf{B}^2$, $i \in [1 : l]$, and reduces them to one output tupel $r^t := (r, r')$ such that: $r^t \in TR_1 \Longleftrightarrow (b_{l-1}^t, \ldots, b_0^t) \in TR_l$.

The standard realization of a two-rail checker is as a tree of basic cells trc reducing two bit tupels to one [RF89], i.e. 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 built of basic cells trc is given in Figure 12. 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.)

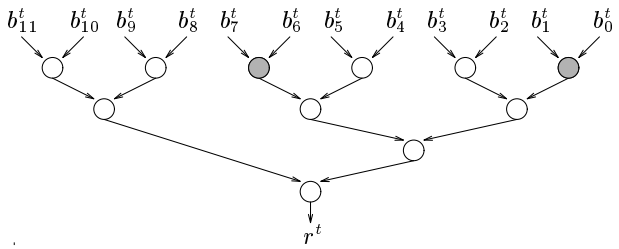

Figure 12: Realization of 12 input two-rail checker

It is well known [RF89] that for the realization of a two-rail checker C as sketched above it holds that:

Lemma 5 *C is self-testing (with respect to stuck-at faults[5]) 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

[5]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.

(see Figure 10), 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 10. By Equations 4 and 5 we can determine the set of normal outputs for the basic cells of a recoder (see Figure 9).

(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.:

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

(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:

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

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 12 for checking the duplicated Booth recoder of Figure 9, 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.

This problem can be solved by observing that the output tupels 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 13 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.

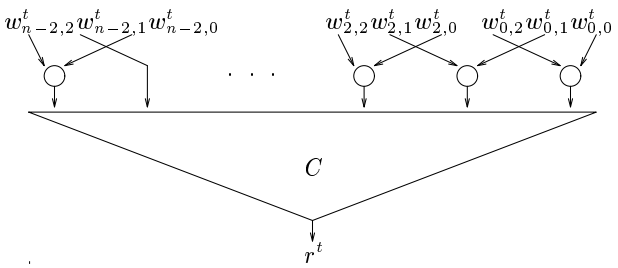

Figure 13: Self-testing two-rail checker

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 13.*

Proof: (Sketch)
Let us first consider the *trc*-cells of the first level in Figure 13 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 [RF89], and can be found in [SR93]. ■

Residue checker There are a lot of different possibilities for designing residue checkers (see for example [AR77, NPP88, Pie91a, Pie91b]). In the following, we will assume that $b = 7$, and the residue checker is built as shown in Figure 8 with the $\mathrm{mod}_{TC}\, b$ module realized as a tree of 3-bit adders with end-around carry (for details see [SR93]). At the output of the tree a circuit is added which maps 111 to 000 in order to have a singular representation of zero.

The set of normal inputs for the two-rail checker in Figure 8 is $\{0^*, 1^*\}^3 \setminus \{1^*1^*1^*\}$. Applying Lemma 5, it can be easily concluded that the circuit is self-testing. For the residue computation tree we already noted in Example 2 that not all possible input combinations are applicable. Thus, care must be taken for structuring this circuit. Again (as for the two-rail checker monitoring the results of Booth recoding) the problem caused by the incomplete input space can be solved by permuting the inputs to module $\mathrm{mod}_{TC}\, b$ appropriately. The permutation can be restricted to the leading bit positions, since for the least significant positions all input combinations are applicable. (The number representations corresponding to $I(p) \in [0 : 2^{2n-2}]$ include the set $\{00w | w \in \mathbf{B}^{2n-2}\}$.) For brevity, we omit the details which can be found in [SR93].

6 Overhead estimation

We 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). 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.

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 with (without) 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) Booth rec.		(2) no Booth rec.	
width	lat.	no lat.	lat.	no lat.
16	22.2 %	42.8 %	18.4 %	31.1 %
32	9.9 %	20.7 %	7.8 %	14.1 %
64	4.5 %	10 %	3.5 %	6.6 %
128	2.2 %	5 %	1.7 %	3.2 %

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 size 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.

References

[AR77] M.J. Ashjaee and S.M. Reddy. On totally self-checking checkers for seperable codes. *IEEE TOC*, C-26:737–744, August 1977.

[Avi71] A. Avizienis. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. *IEEE TOC*, C-20:1322–1331, 1971.

[BI88] T.J. Brosnan and N.R. Strader II. Modular error detection for bit-serial multiplication. *IEEE TOC*, C-37:1043–1052, September 1988.

[BW73] C.R. Baugh and B.A. Wooley. A two's complement parallel array multiplication algorithm. *IEEE TOC*, C-22:1045–1047, December 1973.

[DMD+92] 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. *IEEE VLSI Test Symposium*, pages 280–284, 1992.

[FA92] J. Fadavi-Ardekani. m×n booth encoded multiplier generator using optimized wallace trees. *IEEE Int. Conf. on Comp. Design*, pages 114–117, 1992.

[GG93] M. Gössel and S. Graf. *Error Detection Circuits.* McGraw-Hill, 1993.

[HB81] R.W. Heckelman and D.K. Bhavsar. Self-testing VLSI. *IEEE International Solid-State Circuits Conference*, pages 174–175, 1981.

[HS92] Y.-M. Hsu and E.E. Swartzlander,Jr.. Time redundant error correcting adders and multipliers. *IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 247–256, 1992.

[Kor93] I. Koren. *Computer Arithmetic Algorithms.* Prentice-Hall, 1993.

[LTR89] J.-C. Lo, S. Thanawastien, and T.R.N. Rao. Concurrent error detection in arithmetic and logical operations using berger codes. *9th Symposium on Computer Arithmetic*, pages 233–240, 1989.

[LV83] W.K. Luk and J. Vuillemin. Recursive implementation of optimal time VLSI integer multipliers. *IFIP Congress 83*, pages 155–168, Amsterdam, 1983.

[MRF93] S. Manich, C. Ruiz, and J. Figueras. A fault-secure modified booth multiplier. *EDAC-EUROASIC 93*, pages 180–183, 1993.

[Nic93] M. Nicolaidis. Efficient implementation of self-checking adders and ALUs. *23nd Int. Symp. on Fault-Tolerant Comp. Syst.*, pages 586–595, 1993.

[Nic94] M. Nicolaidis, and H. Bederr. Efficient implementations of self-checking multiply and divide arrays. *The European Design and Test Conference*, pages 574–579, 1994.

[NPP88] D. Nikolos, A.M. Paschalis, and G. Philokyprou. Efficient design of totally self-checking checkers for all low-cost arithmetic codes. *IEEE TOC*, C-37:807–814, July 1988.

[PF83] J.H. Patel and L.Y. Fung. Concurrent error detection in multiply and divide arrays. *IEEE TOC*, C-32:417–422, April 1983.

[Pie91a] S.J. Piestrak. Design of residue generators and multi-operand adders using carry-save adders. *10th Symp. on Comp. Arithm.*, pages 100–107, 1991.

[Pie91b] S.J. Piestrak. Self-testing checkers for arithmetic codes with any check base A. *Pacific Rim Int. Symp. on Fault Tolerant Systems*, pages 162–167, 1991.

[PSSA88] K.F. Pang, H.W. Soong, R. Sexton, and P.H. Ang. Generation of high speed CMOS multiplier-accumulators. *IEEE International Conference on Computer Design*, pages 217–220, 1988.

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

[RF89] T.R.N. Rao and E. Fujiwara. *Error-Control Coding for Computer Systems.* Prentice-Hall, 1989.

[SHB68] F.F. Sellers, M.Y. Hsiao, and L.W. Bearnson. *Error Detecting Logic for Digital Computers.* McGraw-Hill, 1968.

[Span81] O. Spaniol. *Computer arithmetic: logic and design.* Wiley, 1981.

[Spa92] U. Sparmann. On-line error detection in fast adders by residue codes. Technical report, SFB 124, Fachbereich Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, FRG, 1992.

[Spa93] U. Sparmann. On the check base selection problem for fast adders. *IEEE VLSI Test Symposium*, pages 62–65, 1993.

[SR93] U. Sparmann and S.M. Reddy. On the effectiveness of residue code checking for parallel two's complement multipliers. TR 8-21-93, Elec. and Comp. Eng. Dep., University of Iowa, Iowa City, Iowa 52242, 1993.

[Sta89] R. Stans. The testability of a modified booth multiplier. *European Test Conference*, 1989.

[SV91] T.J. Slegel and R.J. Veracca. Design and performance of the IBM Enterprise System/9000 Type 9121 vector facility. *IBM J. Res. Dev.*, 35:367–381, May 1991.

[Tho86] 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.

[TY87] N. Takagi and S. Yajima. On-line error-detectable high-speed multiplier using redundant binary representation and three-rail logic. *IEEE TOC*, C-36:1310–1317, November 1987.

[TY88] N. Takagi and S. Yajima. An on-line error-detectable array divider with a redundant binary representation and a residue code. *18th Fault-Tolerant Computing Symposium*, pages 174–179, 1988.

[Wak78] J. Wakerly. *Error Detecting Codes, Self-Checking Circuits and Applications.* North Holland, 1978.

[WH88] J.W. Watterson and J.J. Hallenbeck. Modulo 3 residue checker: New results on performance and cost. *IEEE TOC*, C-37:608–612, May 1988.

[WT90] B.W.Y. Wei and C.D. Thompson. Area-time optimal adder design. *IEEE TOC*, 39:666–675, 1990.

[ZM86] 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.