Automated Test Generation for Debugging Multiple Bugs in Arithmetic Circuits

Farimah Farahmandi, Member, IEEE, and Prabhat Mishra, Member, IEEE,

Abstract—Optimized and custom arithmetic circuits are widely used in embedded systems such as multimedia applications, cryptography systems, signal processing and console games. Debugging of arithmetic circuits is a challenge due to increasing complexity coupled with non-standard implementations. Existing algebraic rewriting techniques produce a remainder to indicate the presence of a potential bug. However, bug localization remains a major bottleneck. Simulation-based validation using random or constrained-random tests are not effective for complex arithmetic circuits due to bit-blasting. In this paper, we present an automated test generation and bug localization technique for debugging arithmetic circuits. This paper makes four important contributions. We propose an automated approach for generating directed tests by suitable assignments of input variables to make the remainder non-zero. The generated tests are guaranteed to activate bugs. We also propose an automatic bug fixing technique by utilizing the patterns of the remainder terms as well as by analyzing the regions activated by the generated tests to detect and correct the error(s). We also propose an efficient debugging algorithm that can handle multiple independent as well as dependent bugs. Finally, our proposed framework, consisting of directed test generation, bug localization and bug correction, is fully automated. In other words, our framework is capable of producing a corrected implementation of arithmetic circuits without any manual intervention. Our experimental results demonstrate that the proposed approach can be used for automated debugging of large and complex arithmetic circuits.

Index Terms—Algebraic Rewriting, Directed Test Generation, Remainder-based Debugging, Bug Localization, Error Correction, Multiple Bugs Correction.

1 INTRODUCTION

Increasing complexity of integrated circuits increases the probability of bugs in designs. To make it worse, the reduction of time to market puts a lot of pressure on verification and debug engineers to potentially faulty sign-off. The situation gets further exacerbated for arithmetic circuits as the bit-blasting is a serious limitation for most of the existing validation approaches [1]. Faster bug localization is one of the most important steps in design validation.

The urge of high speed and high precision computations increases use of arithmetic circuits in real-time applications such as multimedia and cryptography operations. Moreover, ensuring the security of hardware circuits demands fast and precise arithmetic components. Optimized and custom arithmetic architectures are required to meet the high speed and precision constraints. There is a critical need for efficient arithmetic circuit verification and debugging techniques due to error-proneness of non-standard arithmetic circuit implementations [2]. Recent algebraic rewriting techniques have automated the verification of arithmetic circuits; however, debugging and bug localization still suffer from many manual interactions. Hence, automated debugging of arithmetic circuits is absolutely necessary for efficient design validation.

A major problem with design validation is that we do not know whether a bug exists, and how to quickly find and fix it. Moreover, we do not know how many bugs exist in the design. We can always keep on generating random tests, in the hope of activating the bug(s); however, random test generation is neither scalable nor efficient when designs are large and complex. Existing directed test generation techniques [3], [4] are promising only when the list of faults (bugs) is available. However, they are not applicable when bugs are not known. We propose a directed test generation technique that is guaranteed to activate multiple bugs (if any). The generated tests would also help for faster bug localization.

Existing arithmetic circuits verification approaches have focused on checking the equivalence between the specification of a circuit and its implementation. They use an algebraic model of the implementation [1], [5], [6], [7] using a set of polynomials \( F \). The specification of an arithmetic circuit can be modeled as a polynomial \( f_{spec} \) using a numerical representation of primary inputs and primary outputs. The verification problem is formulated as mathematical manipulation of \( f_{spec} \) over polynomials in \( F \). If the gate-level netlist has correctly implemented the specification, the result of the algebraic rewriting is a zero polynomial; otherwise, it produces a non-zero polynomial containing primary inputs as variables. We call this polynomial remainder. Remainder generation is one-time effort and multiple counterexamples (directed tests) can be generated from one remainder. Any assignment to remainder’s variables that make the remainder to have a non-zero numerical value, generates one counterexample. There can be several possible assignments that make remainder non-zero; each of these assignments is essentially a test vector that is guaranteed to activate at least one of the existing bugs in the implementation.

In this paper, we present a framework for directed test generation and automated debugging of datapath intensive applications using the remainder to locate and correct the bugs in the implementation. Fig. 1 shows an overview of
our proposed framework. Our method generates directed test vectors that are guaranteed to activate the bug. We consider gate misplacement or signal inversion that change the functionality of the design as our fault model. Next, we apply the generated tests, one by one, to find the faulty outputs that are affected by the existing bug. Regions that contribute in producing faulty outputs as well as their intersections are utilized for faster bug localization. We show that certain bugs manifest specific patterns in the remainder. This observation enables automated debugging to find and correct the source of error. We have applied our method on large combinational arithmetic circuits (including 256-bit multipliers) to demonstrate the usefulness of our approach.

![Diagram](https://example.com/diagram.png)

**Figure 2.** Illustrative faulty scenarios for a given design with bug-specific input and output cones. Each star represents one bug. Here, PI and PO refer to the primary inputs and primary outputs, respectively.

Figure 2 shows different scenarios of a buggy implementation. Figure 2 (a) illustrates the case when only one bug exists in the implementation. Figure 2(b) shows the presence of two bugs which do not share input cones (independent bugs). We describe how to fix one or more independent bugs in Section 4 and Section 5.1, respectively. We refer to “activation independence” in the context of activating independent bugs as shown in Figure 2 (b). In other words, independent bugs do not have any overlapping input cones. On the other hand, their effect can be seen in different primary outputs. The effect of different bugs (dependent and independent bugs) may appear in overlapping or non-overlapping cones as shown in Figure 2. We present algorithms to locate and correct multiple independent bugs. In many cases, bugs may share input cones (dependent bugs) as shown in Figure 2 (c). In this paper, we also propose an algorithm to automatically fix dependent bugs in Section 5.2. Generally, a buggy implementation can contain any combination of independent and dependent bugs as shown in Figure 2 (d).

Figure 3 shows different steps of our proposed debugging approach to locate and correct multiple bugs for various scenarios depicted in Figure 2. The existence of a non-zero remainder as a result of applying the functional verification between specification and implementation of an arithmetic circuit is a sign of a faulty implementation. However, there is no information about the number of existing bugs in the implementation. There can be a single bug or multiple independent/dependent bugs in the design. In Section 4, we present a single bug correction algorithm. The main question is that how to know the number of remaining bugs in the design and which algorithm should be used to fix them. In order to determine that whether there is more than one bug in the implementation, we try to partition the remainder $R$ into sub-remainders $R_i$ first. If the remainder can be partitioned successfully into $n$ sub-remainders, we can conclude that there are at least $n$ independent bugs in the implementation as we discussed in Section 5.1. Algorithms in Section 4 are used over each sub-remainder $R_i$ to fix each bug. However, if a single bug cannot be found for remainder $R_i$, there are multiple dependent bugs which construct the sub-remainder $R_i$. Therefore, we try to find a single bug corresponding to remainder $R_i$ first. If we can find such a bug, the bug will be fixed. Otherwise, we try the proposed algorithm of Section 5.2 to find dependent bugs responsible for sub-remainder $R_i$, The procedure is repeated for all of the sub-remainders. To the best of our knowledge, our proposed method is the first attempt to automatically correct multiple dependent/independent bugs in arithmetic circuits.

The remainder of the paper is organized as follows. We discuss related work in Section 2. Section 3 gives an overview about the algebraic rewriting and remainder generation. Section 4 discusses our framework for directed test generation and bug localization for a single bug. Section 5 describes our debugging approach to detect and fix multiple bugs. Section 6 presents our experimental results. Finally, Section 7 concludes the paper.

## 2 Related Work

Test generation is extremely important for functional validation of integrated circuits. A good set of tests can facilitate the debugging and help the verification engineer to find the source of problems. Test generation techniques can be classified into three different categories: random, constrained-random [8] and directed [4], [9], [10], [11]. Random test generators are used to activate unknown errors; however, random test generation is inefficient when designs are large and complex. Constrained-random test generation tries to guide random test generator towards finding test vectors that may activate a set of important functional scenarios. The probabilistic nature of these constraints may lead to situations which can result in generating inefficient tests.
3 BACKGROUND: REMAINDER GENERATION

Several algebraic rewriting approaches have been proposed to verify an arithmetic circuit’s implementation against its specification. A class of these techniques are based on computer symbolic algebra. They map the verification problem as an ideal membership testing [6], [22]. Another class of techniques are based on functional rewriting [1], [23], [24], [25]. These methods can be applied on combinational [5], [26], [27] and sequential [28], [29] Galois Filed $F_{2^n}$ arithmetic circuits using Gröbner Basis theory [30] as well as signed/unsigned integer $\mathbb{Z}_{2^n}$ arithmetic circuits [1], [6], [31], [32], [33].

The specification of an arithmetic circuit can be represented as a word-level polynomial $f_{spec}$. In the specification polynomials, variables are in the symbolic form of the primary inputs and primary outputs of the circuits. A combination of variables as well as integer coefficients constructs the polynomial’s terms. Suppose that we have a multiplier with $\{a_0, a_1, \ldots, a_{n-1}, b_0, b_1, \ldots, b_{m-1}\}$ as primary inputs and $\{z_0, z_1, \ldots, z_{n+m-1}\}$ as primary outputs such that $\{a_i, b_i, z_i\} \subset \mathbb{Z}_2$. The specification of the multiplier can be written as: $\sum_{i=0}^{n+m-1} 2^i z_i - (\sum_{i=0}^{n-1} 2^i a_i, \sum_{i=0}^{m-1} 2^i b_i) = 0$. So, the specification polynomial would be in the following form: $(2^{n+n-1} z_{n+m-1} + \ldots + 2 z_1 + z_0) - (2^{n-1} a_{n-1} + \ldots + 2 a_1 + a_0, 2^{m-1} b_{m-1} + \ldots + 2 b_1 + b_0) = 0$.

To perform verification, the algebraic model of the implementation is used. In other words, each gate in the implementation is modeled as a polynomial with integer coefficients and variables from $\mathbb{Z}_2$ ($x \in 0, 1 \rightarrow x^2 = x$). Variables can be selected from primary inputs/outputs as well as internal signals in the implementation. These polynomials are derived in a way that they describe the functionality of a logic gate. Equation 1 shows the corresponding polynomial of NOT, AND, OR, XOR gates. Note that, any complex gate can be modeled as a combination of these gates and its polynomial can be computed by combining the equations shown in Equation 1.

$$
\begin{align*}
z_1 &= NOT(a) \rightarrow z_1 = -(1-a) = 0, \\
z_2 &= AND(a, b) \rightarrow z_2 = (a \cdot b) = 0, \\
z_3 &= OR(a, b) \rightarrow z_3 = (a + b - a \cdot b) = 0, \\
z_4 &= XOR(a, b) \rightarrow z_4 = (a + b - 2a \cdot b) = 0
\end{align*}
$$

Equation 1

The verification method is based on transforming $f_{spec}$ using information that we directly extract from gate-level implementation. Then, the transformed specification polynomial is checked to see if the equality to zero holds. To fulfill the term substitution, the topological order of the circuit is considered (primary outputs have the highest order and primary inputs are treated as primary outputs).
primary inputs have the lowest). By considering the derived variable ordering, each non-primary input variable which exists in the \( f_{\text{spec}} \) is replaced with its equivalent expression based on its polynomial. Then, the \( f_{\text{spec}} \) is updated and the process is continued on the updated \( f_{\text{spec},n+1} \) until we reach a zero polynomial or a polynomial that only contains primary inputs (remainder). Note that, using a fixed variable (term) ordering to substitute the terms in the \( f_{\text{spec}} \) results in having a unique remainder [30]. Example 1 shows the verification process of a faulty 2-bit multiplier.

Example 1: Consider a 2-bit multiplier with gate-level netlist shown in Fig. 4. Suppose that we deliberately insert a bug in the circuit shown in Fig. 4 by putting the XOR gate with inputs \((A_0, B_0)\) instead of an AND gate. The specification of a 2-bit multiplier is shown by \( f_{\text{spec}} \). The verification process starts from \( f_{\text{spec}} \) and replaces its terms one by one using information derived from the implementation polynomials as shown in Equation 2. For instance, term 4.\(Z_2\) from \( f_{\text{spec}} \) is replaced with \((R + O - 2.R.O)\). The topological order \( \{Z_3, Z_2\} > \{Z_1, R\} > \{Z_0, M, N, O\} > \{A_0, A_1, B_0, B_1\} \) is considered to perform term rewriting. The verification result is shown in Equation 2. Clearly, the remainder is a non-zero polynomial and it reveals the fact that the implementation is buggy.

\[
\begin{align*}
\text{step}_3(\text{remainder}) : & 1.A_0 + 1.B_0 - 3.A_0.B_0
\end{align*}
\]

The existence of a bug in the deeper stages of the design may make it difficult to generate the remainder due to term explosion effect. The reason is the faulty gate may introduce new terms to the intermediate steps of the specification polynomial’s reduction which show the difference of the functionality of the buggy and corresponding correct gate. These extra terms are multiplied to polynomials of other gates and grow continuously until the remainder contains only primary inputs (we call it remainder’s term explosion effect). There are also research efforts [25] that try to reduce the complexity of remainder generation using logic reduction rules during Gröbner basis rewriting. Moreover, word-level abstractions have been exploited for efficient reduction of finite field arithmetic circuits [27]. These approaches enables fast and compact remainder generation. For example, our earlier work [33] has shown that remainders can be generated in a reasonable time (in the order of seconds to few minutes) for complex arithmetic circuits (e.g., 128-bit multipliers) irrespective of the location of the bugs. Specifically, they shown that we can evaluate \( O(n) \) choices (instead of \( 2^n \) in the naïve scenario where \( n \) is the number of inputs in the design) to generate a compact non-zero remainder if the implementation is buggy.

The basic idea of [33] is that more compact remainders can be generated based on partitioning the input space of the design. It is based on applying certain constraints on primary inputs and solve the verification problem for each input constraint. If set \( M = \{0, 1\}^n \) shows all possible input combinations of a design with input bits \( \{x_0, x_1, ..., x_{n-1}\} \) and if specification (S) and implementation (I) are equivalent for all combinations of \( (S \equiv M \subseteq I) \), they should also be equivalent for any input combination that belongs to \( M \). If the implementation is buggy, at least one of the intermediate reductions will result in a non-zero remainder. Therefore, the original verification problem is mapped to \( n \) sub-problems where the specification and implementation polynomials are updated by applying the corresponding constraints. In each sub-problem, the corresponding specification polynomial is reduced over the related implementation polynomials.

Example 2: Assume that we want to partition the input space of the 2-bit multiplier shown in Figure 5. Suppose that primary inputs are given in the following order: \( \{A_1, B_1, A_0, B_0\} \). Table 1 shows the four different constraints on primary inputs. It can be easily verified that these four constraints cover the entire primary inputs’ space. The first and second rows cover two combinations each, the third row covers four combinations, and the last row covers eight combinations. Therefore, it covers all sixteen combinations in Table 1.

![Fig. 4. Faulty gate-level netlist of a 2-bit multiplier](image)

\[
\begin{align*}
\text{TABLE 1} & \\
\text{Input constraints to efficiently verify and debug faulty circuit shown in Figure 5.} & \\
A_1 & B_1 & A_0 & B_0 \\
--- & --- & --- & --- \\
A_1 & 0 & 0 & 0 \\
0 & A_1 & 0 & 0 \\
A_1 & B_1 & 1 & 0 \\
A_1 & B_1 & 1 & 0 \\
\end{align*}
\]

Now, we can apply the incremental algebraic rewriting [33] using all of the input constraints shown in Table 1 to verify the correctness of the implementation. Equation 3 shows the steps of the verification. The specification and implementation polynomials are updated using each constraint. For instance, polynomial of gate 3 is computed as:

\[
\begin{align*}
\text{step}_3(\text{remainder}) : & 1.A_0 + 1.B_0 - 3.A_0.B_0
\end{align*}
\]

$N = A_0 \ast B_1 = 0$ as $A_0$ and $B_1$ are considered zero in the first iteration (first row of the Table 1). Since the last iteration generates a non-zero remainder, the implementation is faulty. The generated remainder also has lower complexity compared to the original remainder that can be obtained using existing methods ($R = 8 \ast A_1 \ast B_1 - 8 \ast A_1 \ast A_0 \ast B_0 \ast B_1$).

To make this approach applicable for any bug in the design, we developed an incremental remainder generation based on a dynamic ordering that utilizes binary search. We provide the proof that the incremental approach is beneficial for debugging since it drastically reduces the complexity of the remainder in the presence of a bug, and leads to faster bug localization. In order to show the correctness of a gate-level netlist, we need to perform all of the iterations to conclude that circuit is correct. In other words, if we want to cover about the correct behavior of the circuit implementation, we need to either check all of the iterations (as shown in Theorem 1 in [33]) to make sure the complete input space or keep all the variables in their symbolic form and perform the ideal membership testing. For example, if any of the iterations of the incremental approach generates a zero remainder, we cannot reach the conclusion that the circuit implementation is correct as $F_{spec} \in J_0 + J(C)^{T_i} \neq F_{spec} \in I(C)$. We have shown in the Theorem 1 in [33] that all $n$ rows of input conditions should be considered to cover the whole space of $J_0 + J(C)$ in order to decide the implementation is correct. If the netlist is buggy, one of the iterations is guaranteed to produce a compact remainder. Note that our incremental approach can be performed in a parallel fashion as iterations are independent of each other.

4 Automated Debugging using Remainders

Our framework uses the remainder that is generated by algebraic rewriting technique as discussed in Section 3. If the remainder is a non-zero polynomial, it means that the implementation is buggy; however, the source of the bug is unknown. Our approach takes the remainder and the buggy implementation as inputs and tries to find the source of error in the implementation and correct it. As shown in Fig. 1, our debugging framework has three important steps. First, we use the remainder to generate directed tests to activate faulty scenarios. Next, we try to localize source of the bug by leveraging the generated tests. Finally, we use an automated correction technique to detect and correct the bug which resides in the suspicious area. We describe each of these steps in detail in the following sections.

4.1 Directed Test Generation

It has been shown that if and only if the remainder is zero, the implementation is bug-free [31]. Thus, when we have a non-zero polynomial as a remainder, any assignment to its variables that makes the numerical value of the remainder non-zero is a bug trace. In our proposed approach, we make use of the remainder to generate test cases to activate unknown bugs. The test is guaranteed to activate the bug in the design. The remainder is a polynomial with Boolean/integer coefficients. It contains a subset of primary inputs as its variables. Our approach takes the remainder and finds the possible assignments to its variables such that it makes the numerical value of the remainder non-zero.

As shown in Example 1, the remainder may not contain
all of the primary inputs. As a result, our approach may use a subset of the primary inputs (that appear in the remainder) to generate directed tests with don’t cares. Such assignments can be found using a SMT solver by defining Boolean variables and considering signed/unsigned integer values as the total value of the remainder polynomial \( (i \neq 0 \in \mathbb{Z}, \text{check}(R = i)) \). The problem of using SMT solver is that for each \( i \), it finds at most one assignment of the remainder variables to produce value of \( i \), if possible. We implemented an optimized algorithm to find all possible assignments that produce non-zero numerical values of the remainder. Algorithm 1 shows the details of our test generation method. The algorithm takes remainder \( R \) polynomial as well as primary inputs (PI) as inputs and generates a set of directed tests \( \mathbb{T} \) to activate the bug. A remainder is constructed as a set of terms as \( R = T_1 + T_2 + \ldots + T_n \) where each term \( T_j \) is a product of a coefficient \( C_j \) and a monomial \( M_j \). The algorithm tries different sets of binary values to PIs \( (s_i)s_i \) and computes the numerical value of \( R \) for assignment \( s_i \). \( M_j \) is a product of binary variables. The value of \( M_j \) is either one or zero as it is a product of some binary variables (line 7). Therefore, the term value may be zero or equal to the term coefficient (\( C_j \)). To compute the numerical value of \( R \) for assignment \( s_i \), the algorithm computes the sum of the values of all the terms in the remainder (lines 4-8). If the sum of all the terms is non-zero, the corresponding primary input assignments are added to the set of Tests (lines 9-10). The test generation algorithm can be implemented in a parallel fashion to improve its performance.

Algorithm 1 Directed Test Generation Algorithm

1: procedure Test–Generation
2: Input: Remainder, R
3: Output: Directed Tests \( \mathbb{T} \)
4: for different assignments \( s_i \) of PIs in R do
5: \( \text{Sum} = 0 \)
6: for each term \( T_j = C_j, M_j \in R \) do
7: \( \text{if} \ (M_j(s_i) \neq 0) \) then
8: \( \text{Sum} + = C_j \)
9: \( \text{if} \ (\text{Sum} != 0) \) then
10: \( \mathbb{T} = \mathbb{T} \cup s_i \)
11: return \( \mathbb{T} \)

Example 3: Consider the faulty circuit shown in Fig. 4 and the remainder polynomial \( R = A_0 + B_0 - 3.A_0.B_0 \). The assignments that make \( R \) to have a non-zero numerical value (\( R = 1 \) or \( R = -1 \)) are \( (A_0 = 1, B_0 = 0), (A_0 = 0, B_0 = 1) \) and \( (A_0 = 1, B_0 = 1) \). These are the scenarios that make difference between functionality of an AND gate and an XOR gate. Otherwise, the fault will be masked since when \( (A_0 = 0, B_0 = 0) \), AND and XOR produce the same output.

### Table 2
Directed tests to activate fault shown in Fig. 4

<table>
<thead>
<tr>
<th>( A_0 )</th>
<th>( A_0 )</th>
<th>( B_0 )</th>
<th>( B_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>

4.2 Bug Localization

So far, we know that the implementation is buggy and we have all the necessary tests to activate the faulty scenarios. Our goal is to reduce the state space in order to localize the error by using the tests generated in the previous section. The bug location can be traced by observing the fact that the outputs can possibly be affected by the existing bug. We apply the tests, simulate the circuit and compare the outputs with the golden outputs (golden outputs can be found from the specification polynomials) and keep track of faulty outputs in set \( E = \{ e_1, e_2, \ldots, e_n \} \). Each \( e_i \) denotes one of the erroneous outputs. To localize the bug, we partition the gate-level netlist to find fanout-free cones (set of gates that are directly connected together) of the implementation.

Each gate whose output is connected to more than one gate is selected as a fanout. For generality, gates that produce primary outputs are also considered as fanouts. To partition implementation, gate-level netlist as well as a list of fanouts \( (L_{fo}) \) are taken. In each iteration, one fanout-gate is chosen from list \( L_{fo} \) and gate level netlist is traced backward until the gate \( g_i \) is reached. The inputs of \( g_i \) can come from one of the fanouts in the list \( L_{fo} \) or primary inputs. All of the visited gates are marked as one cone. This process continues until all of the fanouts are visited.

Algorithm 2 shows the bug localization procedure. Given a partitioned erroneous circuit and a set of faulty outputs \( E \), the goal of the automatic bug localization is to identify all of the potentially responsible cones for the error. First, we find a set of cones \( C_{e_i} = \{ e_1, e_2, \ldots, e_n \} \) that constructs the value of each \( e_i \) from set \( E \) (line 4-5). These cones contain suspicious gates. We intersect all of the suspicious cones \( C_{e_i} \) to prune the search space and improve the efficiency of bug localization algorithm. The intersection of these cones are stored in \( C_S \) (line 7-8).

Algorithm 2 Bug Localization Algorithm

1: procedure Bug–Localization
2: Input: Partitioned Netlist, Faulty Outputs \( E \)
3: Output: Suspected Regions \( C_S \)
4: for each faulty output \( e_i \in E \) do
5: find cones that construct \( e_i \) and put in \( C_{e_i} \)
6: \( C_S = C_{e_0} \)
7: for \( e_i \in E \) do
8: \( C_S = C_S \cap C_{e_i} \)
9: return \( C_S \)

When the effect of the bug can be observed in multiple outputs, it means that the location of the bug is in the intersection of cones which constructs the faulty outputs. We use this information to detect and correct the bug. We describe the details of debugging in Section 4.3.

Example 4: Consider the faulty 2-bit multiplier shown in Fig. 6. Suppose the AND gate with inputs \( (M, N) \) has been replaced with an OR gate by mistake. So, the remainder is \( R = 4.A_1.B_0 + 4.A_0.B_1 - 8.A_0.A_1.B_0.B_1 \). The assignments that activate the fault are calculated based on method demonstrated in Section 4.1. Tests are applied and the faulty outputs are obtained as \( E = \{ Z_2, Z_3 \} \). Then, the netlist is partitioned to find fanout free cones. The cones involved in construction of faulty outputs are: \( C_{Z_2} = \{ 2, 3, 4, 6, 7 \} \)
and \( C_{Z_1} = \{2, 3, 4, 6, 8\} \). The intersection of the cones that produce faulty outputs is \( C_S = \{2, 3, 4, 6\} \). As a result, gates \{2, 3, 4, 6\} are potentially responsible as the source of error.

\[ \text{Fig. 6. Faulty gate-level netlist of a 2-bit multiplier with associated tests} \]

### 4.3 Error Correction

After test generation and bug localization, the next step is error detection. The remainder is helpful since it contains valuable information about the nature of the bug and its location. For example, when the faulty gate is located in the first level (inputs of faulty gates are primary inputs), it creates certain patterns in the remainder. These specific patterns are due to the termination of the substitution process in the algebraic rewriting after this level, which prevents errors from propagating any further. In Example 1, the first level XOR gate is placed by mistake instead of an AND gate. Let us consider the effect of the bug from an algebraic point of view: the equivalent algebraic value of \( Z_0 \) is \( M = A_0 + B_0 - 2.A_1.B_0 \) in the erroneous implementation; however, in the correct implementation, \( Z_0 \) should be equal to \( Z_0^* = A_0.B_0 \). Thus, the difference between \( Z_0 \) and \( Z_0^* \), \( (A_0 + B_0 - 3.A_1.B_0) \) will be observed in the remainder. Therefore, whenever \( a + b - 3.a.b \) pattern is seen in the remainder and there is an XOR gate with inputs \((a,b)\) in the implementation, we can conclude that the XOR gate is the source of error and it should be replaced with an AND gate. Table 3 shows the patterns that will be observed for misplacement of different types of gates. Note that, 3-input (or more) gates can be modeled as cascades of 2-input gates. So, the patterns are also valid for complex gates.

From Section 4.2, we have a set of cones \( C_S \) such that their gates are potentially responsible for the bug. First, the gates in \( C_S \) are extracted and they are kept in a set \( G \). Next, the suspicious gates from the first level of \( G \) are considered and the remainder is scanned to check whether one of the patterns in Table 3 is recognized. If the pattern is found, the faulty gate is replaced with the corresponding gate. Otherwise, the terms of the remainder are rewritten such that it contains output variable of first level gates (at this time, we are sure that the first level gates are not the cause of the problem). We also remove the non-faulty gates from \( G \). Then, we repeat the process over the remaining gates in \( G \) until we find the source of the error.

**Example 5:** Consider the faulty circuit shown in Example 4. The remainder is \( R = 4.A_1.B_0 + 4.A_0.B_1 - 8.A_0.A_1.B_0.B_1 \) and the potentially faulty gates are numbered as 2, 3, 4 and 6. As we can see, remainder \( R \) does not contain any patterns shown in Table 3. It means that the first level suspicious gates 2, 3 and 4 are not responsible for the fault. Thus, we try to rewrite the remainder’s terms with the output of the correct gates. In this step, we know that gates 2, 3 and 4 are correct so their algebraic expressions are also true. As 6 is the only remaining gate, it is the answer. However, we continue the process to show the final solution. By considering \( M = A_1.B_0 \) and \( N = A_0.B_1 \), \( R \) will be rewritten as \( R^* = 4.(M + N - 2.M.N) \) (signal’s weight is computed as shown in [21]). Now, we consider the gates in the second level. This time \( R^* \) matches with one of the patterns shown in Table 3. Based on Table 3, an AND gate with \((M,N)\) as its inputs has been replaced with an OR gate. The only gate that has these characteristics is gate 6 which is also in \( G \). It means that the source of the error has to be the gate 6 and if replaced with an AND gate, the bug will be corrected.

Finding and factorizing of remainder terms in order to rewrite them would be complex for larger designs. To overcome the complexity and obviate the need for manual intervention, we propose an automated approach shown in Algorithm 3. The algorithm takes faulty gate-level netlist, remainder \( R \) and potentially faulty gates of set \( G \) (sorted based on their levels) as inputs. It starts from the first level gate \( g_i \); if \( g_i \) is the buggy gate, one of the patterns in Table 3 should have been manifested in the remainder based on \( g_i \)’s type. Therefore, the debugging algorithm computes two patterns \((P_1, P_2)\) with \( g_i \)’s inputs (lines 7-12) and scan the remainder to check whether one of them matches. If one of the patterns is found, the bug is identified and it can be corrected based on Table 3 (lines 13-16). Otherwise, \( g_i \) is correct and it will be removed from set \( G \) and next gate will be selected. Moreover, the current algebraic expression of \( g_i \) is true and it can be used in subsequent iterations (gate \( g_j \) from higher levels gets the output of \( g_i \) as one of its inputs, the expression of \( g_i \) can be used instead of its output variables). Since our goal is to compute patterns such that they contain just primary inputs, we use a dictionary to keep the expression of the gate output based on the primary inputs (line 19). The weight of each gates’ output is computed based on the weight of its inputs. The weights of the primary inputs and primary outputs are known a priori. The weights of any internal signals can be computed recursively utilizing forward as well as backward traversal. We can also utilize the following properties for different gates. For XOR and OR gates, the output’s weight is same as inputs weight. In multipliers, the output’s weight of the first level AND gates is computed as multiplication of inputs’ weights (they are responsible for partial products). On the other hand, the output’s weight of other AND gates in the design is computed as the sum of inputs’ weights (since they are mostly used in half adders [21]). In adders, the output’s weight of all AND gates is computed as union of input’s weights. This process continues until the bug is
detected or set \( G \) is empty. Since, the algorithm starts from primary inputs, it will not reach a gate whose inputs do not exist in the dictionary. Note that, our debugging approach does not need all of the counterexamples to work. It works even if there is no counterexample (all of the gates are considered as suspicious) or there is just one counterexample. However, having more counterexamples improves debug performance.

Algorithm 3 Error Correction

1: procedure BUG-CORRECTION
2: Input: Suspicious gates \( G \), remainder \( R \)
3: Output: Faulty gate and solution
4: sort \( g_i \) based on their levels (lowest level first)
5: for each level \( j \) do
6:   for each \( g_i \in G \) from level \( j \) do
7:     \((a, b) = \text{input}(g_i)\)
8:     if \((a, b) \text{ are from PI}\) then
9:       \(a = \text{dic}.get(a)\)
10:      \(b = \text{dic}.get(b)\)
11:     \(P_1 = \text{Compute}\.P_1(a, b)\)
12:     \(P_2 = \text{Compute}\.P_2(a, b)\)
13:     if \((P_1 \text{ is found in } R)\) then
14:       return gate \( g_i \) and solution \( S_1 \) from Table 3
15:     else if \((P_2 \text{ is found in } R)\) then
16:       return gate \( g_i \) and solution \( S_2 \) from Table 3
17:     else
18:       remove \( g_i \) from \( G \)
19:       dic.add(output(g_i), Expression(g_i(a, b)))

Example 6: We want to apply Algorithm 3 on the case shown in Example 5. We start from gate 2 and compute \( P_1 = -2.A_1 - 2.B_0 + 4.A_1.B_0 \) and \( P_2 = -2.A_1 - 2.B_0 + 6.A_1.B_0 \) for gate 2. As these patterns do not exist in the remainder, gate 2 is correct and the dictionary will be updated as \((M = 2.A_1.B_0)\). The same will happen for gate 3 and 4, and the dictionary will be updated as \((M = 2.A_1.B_0, N = 2.A_0.B_1)\) at the end of this iteration. When we consider gate 6, the \( P_i \)'s are as follows: \( P_1 = 4.A_1.B_0 + 4.A_0.B_1 - 8.A_1.B_0.A_0.B_1 \) and \( P_2 = 4.A_1.B_0.A_0.B_1 \). Considering that \( R = 4.A_1.B_0 + 4.A_0.B_1 - 8.A_0.A_1.B_0.B_1 \), \( P_1 \) of gate 6 can be observed in \( R \). So the bug is the OR gate 6, and based on Table 3 it can be fixed by replacing with an AND gate.

Signal inversion problem can be viewed in the same way as gate replacement error. If we consider a wire as a buffer, it may be replaced with an inverter. Therefore, it is a special class of gate replacement error, where a buffer can be replaced with an inverter, or vice versa. For example, assume that signal \( a \) is inverted by mistake in the actual implementation. Therefore, the difference between the expected behavior and the implementation appears in the remainder by performing the functional rewriting of the specification polynomial. In this case, instead of \( a \) we encounter \( 1 - a \) in the implementation, and the remainder is \( R = 1 - a - a = 1 - 2 \times a \). As a result, the appearance of the pattern \( 1 - 2 \times a \) in the remainder reveals the fact that signal \( a \) is inverted by mistake.

5 Debugging Multiple Bugs

Section 4 presented algorithms for detecting, localizing and correcting a single bug. In this section, we extend these algorithms for debugging multiple errors. The fault model (gate replacement) as well as remainder generation process remains the same. If the algebraic rewriting of an arithmetic circuit results in a non-zero remainder, we know that the implementation is buggy. However, the sources of the errors are unknown. Our plan is to use the non-zero remainder in order to generate directed tests to activate the bugs, localize the source of errors and correct them. First, we explain how we extend the approach presented in Section 4 to correct multiple independent bugs. Then, we present an approach to solve the debugging of two dependent faulty gates.

If there is more than one bug in the implementation, the remainder will be affected by all of them since all of the faulty gates are contributing in the algebraic rewriting procedure as well as the remainder generation. In other words, the remainder shows the effect of all bugs in the implementation. Example 7 shows how the remainder is generated when there are two bugs in the implementation.

Example 7: In the circuit shown in Figure 7, the AND gate with inputs \((A_0, B_0)\) as well as the AND gate with inputs \((A_1, B_1)\) are replaced with XOR and OR gates, respectively (i.e., two faults in the implementation of a 2-bit multiplier). The result of algebraic rewriting (remainder polynomial) can be computed as shown in Equation 4.

\[
step_3(remainder) : R = A_0 + B_0 - 3.A_0.B_0 + 4.A_1 + 4.B_1 - 8.A_1.B_1
\]

(4)

Detailed observation in the remainder generation procedure shows that the overall remainder can be considered as the sum of different individual bug’s effect in the algebraic rewriting process. For instance, one part of the remainder shown in Example 4, comes from the remainder shown in Example 1 (the same bug) as \((A_0 + B_0 - 3.A_0.B_0)\) and the other part \((4.A_1 + 4.B_1 - 8.A_1.B_1)\) is responsible for the second bug and it is equal to the remainder that can be the result of the algebraic rewriting with an implementation which contains only the second bug. Therefore, each assignment that makes the remainder non-zero activates at least one of the existing faulty scenarios. Some tests may activate all of the bugs at the same time. Thus, Algorithm 1 can be used to generate directed tests when there are more than one fault in the design.
Example 8: Directed tests to activate the buggy implementation of Example 7 are shown in Figure 5. The assignments make the first part of the remainder non-zero \((A_0 + B_0 - 3.A_0.B_0)\) and activates the first fault. For example, assignment \((A_1 = 1, A_0 = 0, B_1 = 0, B_0 = 0)\) manifests the effect of the first fault in \(Z_0\). On the other hand, the assignments that make the second part of the remainder non-zero \((4.A_1 + 4.B_1 - 8.A_1.B_1)\), are tests to activate the second bug. Assignment \((A_1 = 1, A_0 = 0, B_1 = 0, B_0 = 0)\) activates the second fault in \(Z_2\). However, the assignment \((A_1 = 1, A_0 = 0, B_1 = 0, B_0 = 1)\) activates both of these faults at the same time \((Z_0\) and \(Z_2))\.

To localize the source of errors, the circuit is simulated using the generated tests to find faulty primary outputs. Faulty gates exist in the cones that construct the functionality of faulty outputs. In order to prune the search space and localize source of errors, we cannot directly apply Algorithm 2 as their intersection may be a zero set. However, some information can be found from using Algorithm 2. In the following sections, we describe the bug localization and correction of multiple bugs: i) Section 5.1 covers bugs with independent input cones (independent bugs), and ii) Section 5.2 covers bugs which share some input cones (dependent bugs).

5.1 Error Correction for Multiple Independent Bugs
We refer two bugs as independent if they have different input cones (fan-ins). Figure 7 shows two independent bugs in a 2-bit multiplier. If multiple bugs are independent of each other, their effect can be observed easily in the remainder as the sum of each individual bug’s remainder (sum of sub-remainders). Therefore, if the remainder is partitioned into multiple sub-remainders based on the primary inputs (each part representing the effect of one bug), each sub-remainder as well as the associate faulty cones can be fed into Algorithm 3 in order to detect and correct the source of multiple independent errors.

If the input cones (input fan-ins) of faulty gates are separate from each other, a different set of primary inputs may appear in each sub-remainder. In order to find the sub-remainders, each term of the overall remainder and its corresponding monomial are examined to determine which sub-remainder it belongs. Algorithm 4 shows the remainder partitioning procedure.

Algorithm 4 takes the overall remainder \(R\) as input and returns the partitioned sub-remainers \(R_i\)s. The algorithm sorts the terms of the \(R\) based on their monomial size (the number of variables in each term) in descending order (line 5). In the next step, it starts from the largest term of the remainder \(R\) and adds it to sub-remainder \(R_0\) (line 6). Then, it examines all terms of \(R\) from the second largest term \(t\) to find out which partition they belong to (lines 7-8). If some of the variables of the term \(t\) already exist in the sub-remainder \(R_i\), the term \(t\) will be added to sub-remainder \(R_i\) (lines 9-10). Otherwise, the algorithm creates a new sub-remainder \(R_j\) and adds \(t\) to it (lines 12-13). The process continues until all terms of the \(R\) are examined. If the algorithm results in only one sub-remainder, it shows that faulty gates do not have independent input cones. The computed sub-remainders are fed into Algorithm 1 in order to generate directed tests activating the corresponding bug of that sub-remainder. The generated tests are used to define the corresponding faulty outputs of each bug. Example 9 illustrates the remainder partitioning procedure.

Example 9: Consider the faulty multiplier design shown in Figure 7 and corresponding remainder shown in Equation 4. In order to find different possible sub-remainders, the remainder is sorted as: \(R = -3.A_0.B_0 - 8.A_1.B_1 + A_0 + B_0 + 4.A_1 + 4.B_1\). The partitioning starts from term \(-3.A_0.B_0\) and as there are no sub-remainder so far, sub remainder \(R_1\) is created and the term is added to it as: \(R_1 = -3.A_0.B_0\). The second term \(-8.A_1.B_1\) is examined and as \(R_1\) does not contain variables \(A_1\) and \(B_1\), new sub-remainder \(R_2\) is created. Similarly, rest of the terms of \(R\) are examined and \(R_1\) and \(R_2\) are computed as: \(R_1 = -3.A_0.B_0 + A_0 + B_0\) and \(R_2 = -8.A_1.B_1 + 4.A_1 + 4.B_1\). The directed tests are shown in Figure 7.

The generated tests are applied and faulty outputs are defined. The faulty outputs of each bug are fed into Algorithm 2 in order to find potential faulty cones. Algorithm 3 is used with each sub-remainder as well as corresponding potential faulty gates as its inputs, and it tries to detect and correct each bug. In other words, the problem of debugging a faulty design with \(n\) independent bugs is mapped to debugging of \(n\) faulty designs where each design contains a single bug. We illustrate how to apply Algorithm 3 to correct multiple independent sources of errors using Example 10.

Example 10: Having the directed tests shown in Figure 7, faulty outputs \(Z_0\) and \(Z_2\) as well as two sub-remainders computed in Example 9, Algorithm 3 is used twice to find the source of errors. In the first attempt, the faulty output is \(Z_0\) and the computed potential faulty cone using Algorithm 2 contains only gate 1. In this gate, gate 1 as well as \(R_1\), are fed into the bug correction algorithm (Algorithm 3). Two patterns \(P_1 = A_0 + B_0 - 3.A_0.B_0\) (if the potential faulty gate 1 should be an AND gate) and \(P_2 = -A_0.B_0\) (if the potential faulty gate 1 should be an OR gate) are computed. Therefore, gate 1 should be replaced with an AND gate to fix the first bug since the \(P_1\) is equal to the remainder \(R_1\). The same procedure is used for the second bug while the potential faulty gates are \(\{2, 3, 4, 6, 7\}\) since the only faulty output is \(Z_2\). Trying different patterns results in a conclusion that gate 4 should be replaced with an AND gate.
5.2 Error Correction for Dependent Bugs

In this section, we describe how to detect and correct dependent bugs that share input cones. The key difference here from the cases that we solved in Section 5.1 is the fact that the remainder cannot easily be partitioned into sub-remainders since some of the terms of the corresponding sub-remainder may be canceled through other sub-remainders or they may be combined to each other. The reason is that the bugs share some input cones (fan-ins) and their individual sub-remainders may have common terms consisting of a set of primary inputs as variables. When sub-remainders are combined to each other to form the overall remainder, some term combinations/cancellations happen. Moreover, some of the sub-remainders may be affected by lower level faults and the presented method in Section 5.1 cannot solve these cases. We illustrate the fact using the following example.

Example 11: Consider the faulty implementation of a 2-bit multiplier with two bugs as shown in Figure 8. Assume that gates 6 and 8 are replaced with OR gates to inject faults. It can be observed from Figure 8 that two bugs share some set of input cones (gates {2, 3, 4} are common in input cones of faulty gates 6 and 7). Applying algebraic rewriting on the circuit shown in Figure 8 results in a non-zero remainder: \( R = 8.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1 \). However, if only gate 6 is replaced with an OR gate in the implementation (single bug), the remainder will be equal to: \( R_1 = 4.A_0.B_1 + 4.A_1.B_0 - 8.A_0.A_1.B_0.B_1 \). Similarly, when only gate 7 is replaced with an OR gate (single fault), the remainder will be computed as: \( R_2 = 8.A_1.B_1 - 8.A_0.A_1.B_0.B_1 \). As it can be observed, \( R \neq R_1 + R_2 \). The reason is that buggy gate 6 has an effect on the generation of sub-remainder \( R_2 \). As a result, \( R_2 \) should be computed as: \( R_2' = 8.A_1.B_1 + 8.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1 + 8.A_0.A_1.B_0.B_1 \). We can observe that \( R = R_1 + R_2' \). Note that there is not any monomial of \( A_0.A_1.B_0.B_1 \) in the remainder \( R \); however, this monomial exists in both \( R_1 \) and \( R_2' \) with opposite coefficients resulting in the term cancellation.

As it can be observed from Example 11, term cancellation as well as lower level bugs’ effect are two main reasons that limit the applicability of the algorithms presented in Section 5.1 to detect and correct bugs with common input cones. In this section, we present a general approach to correct and detect multiple gate misplacement bugs regardless of the bugs’ positions. The first step to fix dependent bugs is to use Algorithm 1 in order to generate directed tests to activate unknown bugs. In the next step, the circuit is simulated using the generated tests to define the faulty outputs (\( E \)) since the effect of faults will be propagated to them. Algorithm 2 cannot be used to localize the potential faulty cones since the intersection of the faulty cones may eliminate some of the faulty gates. Instead, union of all of the gates that construct faulty outputs should be considered as potential faulty gate candidates to make sure that all of the potential faulty gates are considered. The next step is to define faulty gates and their corresponding solutions using the remainder as well as potential faulty gates. We construct two sub-remainders from each potentially faulty gates (e.g., considering if the current gate is faulty and the type of gate is AND, the solution can be either OR gate or XOR gate based on Table 3) and we store them in set \( R \). To be able to detect the bugs, we are looking for \( n \) sub-remainders \( R_i \in R \) where their union construct the original remainder \( R \).

In general, finding \( n \) dependent bugs and constructing the respective remainder \( R \) maps to “subset sum” problem and it has exponential complexity. In other words, we need to find \( n \) potential sub-remainders such that their sum is equal to the remainder \( R \). Therefore, for each gate in a faulty region, we construct two patterns (sub-remainders) as shown in Algorithm 3 as total \( m \) sub-remainders. To be able to detect and correct \( n \) dependent bugs, we need to select \( \binom{m}{n} \) where \( r_1 + r_2 + ... + r_n = R \). The most naïve algorithm to solve this problem is to consider all subset of \( m \) sub-remainders, and check whether the subset sums to \( R \) for every subset. The complexity of this algorithm is in the order of \( O(2^m) \). If we use the naïve approach for finding two dependent bugs, the complexity is \( O(m^2) \). By introducing Algorithm 5 and using a hash map, we could solve this problem in linear time \( O(m) \) for two dependent bugs.

Figure 2 shows all of the buggy scenarios that our current method can automatically debug in linear time. As it can be seen in Figure 2(d), our method is capable of handling \( 2 \times k \) dependent bugs in linear time when we have \( k \) independent faulty regions where each of them has at most two dependent bugs. In other words, Algorithm 4 partitions the remainder \( R \) into \( k \) sub-remainders where for each sub-remainder \( r_{ij} \), our method tries to find at most two dependent bugs in linear time.

To detect two dependent bugs in a faulty region, we are looking for two sub-remainders such that their sum constructs the overall remainder \( R \). Note that sub-remainder of an individual bug may be affected by the other existing bug in the implementation (for instance, sub-remainder \( R_2' \) which shows the effect of faulty gate 7 in Example 11 is also affected by faulty gate 6). Algorithm 5 is used to locate and correct two dependent bugs by finding two sub-remainders \( R_1 \) and \( R_2 \) such that their sum is equal to \( R (R = R_1 + R_2) \). The algorithm tries to find two equal polynomials: \( R = R_1 + R_2 \). The algorithm takes the remainder and potential faulty gates as inputs and it returns two faulty gates and their correct replacement as output. The algorithm consists of three major steps. First, polynomials corresponding to gate’s inputs (we have assumed that a gate has two inputs for simplicity in representation) are computed based on primary inputs for each potentially faulty gate \( g_i \)
and \(b\) are corresponding polynomials of gate \(g_i\) inputs). Computed polynomials are added to map \(\text{dic}\) (lines 6-8). Second, algorithm constructs two patterns \((P_1\) and \(P_2\)) for each potentially faulty gates \(g_i\) based on Table 3 regarding the functionality of their input gates (lines 9-12). Note that \(P_1\) and \(P_2\) can be constructed based on the fact that we have considered only three types of gates (AND, OR, and XOR) so that each gate can be replaced by two other ones. For example, if the suspicious and potentially buggy gate is an AND gate, it can be replaced with either an OR gate or an XOR gate to fix the bug. Therefore, we construct two patterns, one showing the functionality of replacing the AND gate with an OR gate \((P_1)\), and the other one shows the functionality of replacing the AND gate with an XOR gate \((P_2)\). Computed patterns are added to set \(\mathbb{P}\) (line 13).

For computed patterns \(P_1\) and \(P_2\), the algorithm computes the \(R - P_1\) and it stores the result in a map \(\mathbb{R}\) (lines 14-15). In the final step, each of the patterns \(P_j \in \mathbb{P}\) is checked to see whether it exists in the map \(\mathbb{R}\) (lines 16-18). If \(P_j\) exists in map \(\mathbb{R}\), it means that there were a pattern \(P_i\) in set \(\mathbb{P}\) where \(R - P_i = P_j\). Therefore, \(P_1\) and \(P_2\) are the sub-remainderings \(R_1\) and \(R_2\) that we are looking for such that \(R_1 = P_1\) and \(R_2 = P_2\). The gates corresponding to \(P_1\) and \(P_2\) are faulty and their solution can be found based on Table 3 (lines 19-20). Note that, by using hash map \(\mathbb{R}\) the complexity of the algorithm is proportional to the number of faulty gates. The complexity of the algorithm grows linearly with the number of suspicious gates (suspicious gates can be obtained by bug localization phase).

**Algorithm 5 Debugging Two Bugs**

1: procedure DEBUGGING–TWO–DEPENDENT–BUGS
2: Input: Suspicious gates \(\mathbb{G}\), remainder \(R\)
3: Output: Faulty gates and their solution
4: \(\mathbb{P} = \{\} \quad \triangleright \text{A set that keeps patterns for all gates as well as corresponding solution of each pattern}\)
5: \(\mathbb{R} = \{\} \quad \triangleright \text{A map that keeps remainder minus all patterns as well as corresponding patterns}\)
6: for each gate \(g_i \in \mathbb{G}\) do
7: \((a, b) = \text{computeInputPolynomials}(g_i)\)
8: \(\text{dic.add}(g_i, (a, b))\)
9: for each gate \(g_i \in \mathbb{G}\) do
10: \((a, b) = \text{dic.getInputPolynomials}(g_i)\)
11: \(P_1 = \text{computeP1}(a, b)\)
12: \(P_2 = \text{computeP2}(a, b)\)
13: \(\mathbb{P} = \mathbb{P} \cup \{P_1, P_2\}\)
14: \(\mathbb{R}.\text{put}((R - P_1), P_1)\)
15: \(\mathbb{R}.\text{put}((R - P_2), P_2)\)
16: for each \(P_j \in \mathbb{P}\) do
17: if \(P_j\) exists in \(\mathbb{R}\) then
18: \(P_i = \mathbb{R}.\text{get}(P_j)\)
19: \(\text{gate } g_i = \mathbb{R}.\text{get}(P_i)\) is faulty and get solution \(S_i\) from Table 3
20: \(\text{gate } g_j = \mathbb{P}.\text{get}(P_j)\) is faulty and get solution \(S_j\) from Table 3

Note that, Algorithm 5 requires to construct the exact sub-remainder responsible for the potential bugs (it is not useful to find the pattern as some part of the remainder). The exact sub-remainder is dependent on the gates that the buggy gates are connected in the next level of the design. To illustrate the point, suppose that gate \(g_1\) is connected to only a half-adder with inputs \(g_1\) and \(g_2\). If \(f_{g_1}\) and \(f_{g_2}\) show the corresponding polynomials of gates \(g_1\) and \(g_2\) based on the functionality of their inputs, gate \(g_1\) contributes to the functionality of the next level by polynomial

\[
f_{g_1} + f_{g_2} - 2 + f_{g_1} * f_{g_2} = (XOR) + 2 + f_{g_1} * f_{g_2} = (AND) = f_{g_1} + f_{g_2}
\]

However, if the gate \(g_1\) is buggy and its functionality is replaced by polynomial \(f_{a_1}\), there would be a difference in the functionality of the design as: \(\Delta = f_{a_1} - f_{g_1}\). If gate \(g_1\) is connected to a half-adder with inputs \(g_1\) and \(g_2\), the reduction results in \(\Delta + f_{g_2} - 2 + \Delta.f_{g_2} + 2 + \Delta.f_{g_2}\).

Since \(f_{g_2}\) should be included in the correct functionality of the design, the exact sub-remainder can be computed as: \(\Delta - 2 + \Delta.f_{g_2} + 2 + \Delta.f_{g_2} = \Delta\). Patterns that are computed in Table 3, match with \(\Delta\) based on the polynomials of inputs of the faulty gate. In arithmetic circuit implementations, most of the gates are connected to half-adders (or they are in the last level of the design). Therefore, we consider them as potentially faulty gates, their constructed patterns are equal to the exact remainder.

However, if buggy gate \(g_1\) is not connected to a half adder, the exact sub-remainder due to faulty gate \(g_1\) may include more terms besides the terms appears in the \(\Delta\). For example, if \(g_1\) is connected to an XOR gate \(g_2\), the exact remainder would be equal to: \(\Delta - 2 + \Delta.f_{g_2}\). The extra part \(-2 + \Delta.f_{g_2}\) comes from vanishing monomial propagated to the remainder due to the effect of the bug and no counter-part monomials will appear to cancel them during backward algebraic rewriting.

**Example 12:** Consider the faulty full-adder shown in Figure 9. The gate \(G_2\) has been replaced by an OR gate to inject a fault in the gate-level implementation. After the verification procedure, the remainder is: \(R = 2 + (A + B - 2 * A * B) - 2 * C_{in} * (A + B - 2 * A * B)\). The remainder \(R\) has two parts: the first part shows the difference of the functionality of the faulty gate (OR) and the correct gate (AND) as: \(\Delta = (A + B - A * B) - A * B = A + B - 2 * A * B\). However, the second part \((-2 - C_{in} * (A + B - 2 * A * B))\) represents the vanishing monomials propagated to the remainder due to the bug.

![Fig. 9. Faulty netlist with one bug (gate \(G_2\) should have been an AND gate.)](image-url)

In order to construct the exact remainder for a suspicious gate \(g_1\), we construct \(\Delta\) patterns based on Table 3. In the second step, we consider each gate \(g_j\) such that \(g_j\) is its input and we compute the corresponding polynomial \(g_j\) based on its inputs’ polynomials. The terms that contain \(\Delta\)
should be added to the remainder. Note that, if we have two cascaded bugs, the effect mentioned above only happen for the higher level bug since the effect of the lower level bug is considered while constructing the pattern of the higher level bug. Another important aspect is that the weight of each gate should be considered as described in Section 4.3.

**Example 13:** Consider the faulty full-adder shown in Figure 9 where gate $G_2$ has been replaced by an OR gate (it should be an AND gate in the correct implementation). We know that the remainder is equal to $R = 2\ast((A + B - 2\ast A\ast B) - C_{in}\ast(A + B - 2\ast A\ast B))$ and the implementation is buggy. If we are suspicious about the $G_2$ and we guess that it should be an AND gate, we construct $\Delta = A + B - 2\ast A\ast B$ based on Table 3. Since $G_2$ is only input of gate $G_3$, we construct the polynomial as

$$f_{G_4} + \Delta - \Delta \ast f_{G_4}$$

Since the term $f_{G_4}$ is not dependent on $\Delta$, it is a part of the correct functionality of the implementation, and it should not be considered in the remainder. Therefore the constructed remainder is

$$R' = \Delta - \Delta \ast f_{G_4} = 2\ast((A + B - 2\ast A\ast B) - C_{in}\ast(A + B - 2\ast A\ast B))$$

As $R = R'$, we can conclude that gate $G_2$ is buggy and it should be replaced by an AND gate to fix the error. We show how exact sub-remainder are used to debug two dependent bugs in Example 14.

**Example 14:** Consider the faulty implementation of a 2-bit multiplier shown in Figure 8 with remainder: $R = 8\ast A_1\ast B_1 + 12\ast A_1\ast B_0 + 12\ast A_0\ast B_1 - 16\ast A_0\ast A_1\ast B_0 - 16\ast A_1\ast B_0\ast B_1$. Corresponding directed tests to activate existing bugs are shown in Figure 8. Potential faulty gates are computed based on faulty outputs $Z_2$ and $Z_3$ as gates $\{2, 3, 4, 6, 7, 8\}$. Algorithm 5 creates two patterns for each of the suspicious gates as shown in Table 4 column “Pattern”. For each pattern, the possible solution as well as remainder minus patterns are listed in the third and fourth columns of Table 4, respectively. Note that, Table 4 is the combination of two lists, $\mathbb{P}$ and hash map $\mathbb{R}$, which are mentioned in Algorithm 5. Each pattern listed in the second column is tested to find whether it exists in hash map $\mathbb{R}$ (part of hash map is shown in the fourth column). As it can be seen in the table, $P_{11}$ (highlighted polynomial in the second column) is equal to $R - P_2$ (highlighted in the fourth column). It means that $R - P_2 = P_{10} \rightarrow R = P_{11} + P_2$. Therefore, gate 6 and 8 are faulty since $P_7$ and $P_{11}$ are corresponding to these gates and they should be substituted with AND gates.

## 6 Experiments

### 6.1 Experimental Setup

The directed test generation, bug localization, and bug correction algorithms were implemented in a Java program and experiments were conducted on a Windows PC with Intel Xeon Processor and 16 GB memory. We have tested our approach on both pre- [1] and post-synthesized gate-level arithmetic circuits that implement adders and multipliers. Post-synthesized designs were obtained by synthesizing the high-level description of arithmetic circuits using Xilinx synthesis tool. We consider wrong gate (gate replacement bug) or signal inversion which change the functionality of the design as our fault model. Several gates from different levels were replaced with an erroneous gate in order to generate faulty implementations. The remainders were generated based on the method presented in [33]. Multiple counterexamples (directed tests) are generated based on one remainder. As each counterexample can be generated independent of others, so we used a parallelized version of the algorithm for faster test generation. We compared our test generation method with existing directed test generation method [3] as well as random test generation. We have inserted several bugs in the middle levels of the circuits to conduct our experimental results. We compared our debugging results with most recent work in this context [21]. We use the benchmarks obtained from the authors [21]. However, we have implemented their algorithm to compare our method with their method. To enable fair comparison, similar to [21], we randomly inserted bugs (gate changes) in the middle stages of the circuits. We improved the run-time complexity of presented method in [35] by using efficient data structures such as hash maps and sorted sets.

### 6.2 Debugging a Single Error

Table 5 presents results for test generation, bug localization and debugging methods using multipliers and adders. The first column (“Type”) indicates the types of benchmarks. The second (“Size”) and third (“#Gates”) columns show the size of operands and number of gates in each design, respectively. Since the sizes of adder designs are smaller than multiplier designs, we show results only for higher operand sizes (bit-widths). The fourth column (“RG(s)”) shows the CPU time to generate the remainder. The fifth column (“Dir. (s)”) indicates results for directed test generation method presented in [3] by using SMV model checker [36] (We give the model checker the advantage of knowing the bug). The sixth column (“Random (s)”) represents results of random test generation method (time to generate the first counterexample using the random technique). The seventh column (“Our TG (s)”) represents the time of our test generation method that generates multiple tests. As it can be observed from Table 5, our method has improved directed test generation time by several orders of magnitude. The eighth column (“Bug Loc. (s)”) shows the CPU time for bug localization algorithm. The ninth column (“[21](s)”) shows the debugging time of [21] using our implementation in Java. The next column (“Our (TG+BL+DC) (s)”) provides CPU time of our proposed approach which is the sum of test generation (TG), bug localization (BL) and debugging/correction (DC) time. The last column (“Improvement”) shows the improvement provided by our debugging framework. Clearly, our approach is an order-of-magnitude faster than the most closely related approach [21], especially for larger designs as bug localization has an important effect. The reported numbers are the average of generated results for several different scenarios. For instance, if we zoom into test generation of the first row (post-synthesized multiplier with 4-bit operands) of Table 5, the reported results are the average of the nine possible scenarios shown in Table 6.

Table 6 presents the debugging results of 4-bit post-synthesized multiplier. The first column (“Faults”) shows a
TABLE 4
Patterns for potential faulty gates Example 14

<table>
<thead>
<tr>
<th>Gate#</th>
<th>Pattern</th>
<th>Solution</th>
<th>Remainder pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 (AND)</td>
<td>$2A_2 + 2B_2 + 4A_1B_1$</td>
<td>$0A_2 - 2A_1 - 2B_1 + 8A_1B_1 + 16A_2B_1 + 32A_1B_1$</td>
<td>$2B_1 - 2B_2 - 2A_2 + 8A_2B_2$</td>
</tr>
<tr>
<td>3 (AND)</td>
<td>$2A_2 + 2B_2 + 4A_1B_1$</td>
<td>$0A_2 - 2A_1 - 2B_1 + 8A_1B_1 + 16A_2B_1 + 32A_1B_1$</td>
<td>$2B_1 - 2B_2 - 2A_2 + 8A_2B_2$</td>
</tr>
<tr>
<td>4 (AND)</td>
<td>$4A_1B_1 + 8A_2B_2 - 8A_1B_1 + 4A_1B_1$</td>
<td>$2A_2 + 2B_2 + 4A_1B_1 + 8A_2B_2 + 32A_1B_1$</td>
<td>$8A_2B_2 + 4A_1B_1 - 8A_1B_1 - 4A_1B_1$</td>
</tr>
<tr>
<td>6 (OR)</td>
<td>$4A_1B_1 + 8A_2B_2 - 8A_1B_1 + 4A_1B_1$</td>
<td>$2A_2 + 2B_2 + 4A_1B_1 + 8A_2B_2 + 32A_1B_1$</td>
<td>$8A_2B_2 + 4A_1B_1 - 8A_1B_1 - 4A_1B_1$</td>
</tr>
<tr>
<td>7 (OR)</td>
<td>$4A_1B_1 + 8A_2B_2 - 8A_1B_1 + 4A_1B_1$</td>
<td>$2A_2 + 2B_2 + 4A_1B_1 + 8A_2B_2 + 32A_1B_1$</td>
<td>$8A_2B_2 + 4A_1B_1 - 8A_1B_1 - 4A_1B_1$</td>
</tr>
<tr>
<td>8 (OR)</td>
<td>$8A_1B_1 + 8A_2B_2 - 8A_1B_1 + 4A_1B_1$</td>
<td>$2A_2 + 2B_2 + 4A_1B_1 + 8A_2B_2 + 32A_1B_1$</td>
<td>$8A_2B_2 + 4A_1B_1 - 8A_1B_1 - 4A_1B_1$</td>
</tr>
</tbody>
</table>

TABLE 5
CPU time and memory results of debugging arithmetic circuits. Single bug inserted in the middle stages of the design as well as to primary inputs. TO = timeout after 3600 sec; MO = memory out of 8 GB

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type</th>
<th>Verification</th>
<th>Test Generation (TO)</th>
<th>Bug Localization (BL)</th>
<th>Debugging/Correction (DC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>post-syn. Multipliers</td>
<td>4</td>
<td>73</td>
<td>0.03</td>
<td>1.27</td>
<td>0.01</td>
</tr>
<tr>
<td>pre-syn. Multipliers</td>
<td>4</td>
<td>94</td>
<td>0.05</td>
<td>1.27</td>
<td>0.01</td>
</tr>
<tr>
<td>post-syn. Adder</td>
<td>4</td>
<td>94</td>
<td>0.05</td>
<td>1.27</td>
<td>0.01</td>
</tr>
<tr>
<td>pre-syn. Adder</td>
<td>4</td>
<td>94</td>
<td>0.05</td>
<td>1.27</td>
<td>0.01</td>
</tr>
</tbody>
</table>

TABLE 6
Test Generation time for 4-bit multiplier with 8 bits outputs # Gates = 72

<table>
<thead>
<tr>
<th>Faults</th>
<th>#Gates</th>
<th>4-bit</th>
<th>8-bits</th>
<th>Faulty Outputs</th>
<th># Ran. Tests</th>
<th>TO (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR → AND</td>
<td>4</td>
<td>2</td>
<td>0.03</td>
<td>0.05</td>
<td>128.12</td>
<td>1.21</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>3</td>
<td>2</td>
<td>0.03</td>
<td>0.05</td>
<td>128.12</td>
<td>1.21</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>2</td>
<td>2</td>
<td>0.03</td>
<td>0.05</td>
<td>128.12</td>
<td>1.21</td>
</tr>
<tr>
<td>2 XOR → 1 AND</td>
<td>1</td>
<td>2</td>
<td>0.03</td>
<td>0.05</td>
<td>128.12</td>
<td>1.21</td>
</tr>
<tr>
<td>Average</td>
<td>1</td>
<td>2</td>
<td>0.03</td>
<td>0.05</td>
<td>128.12</td>
<td>1.21</td>
</tr>
</tbody>
</table>

The experimental results demonstrate three important aspects of our approach. First, our test generation method generates multiple directed tests when the bug is unknown in a cost-effective way. Second, our debugging approach detects and corrects single fault caused by gate replacement in a reasonable time. Finally, our debugging method is not dependent on any specific architecture of arithmetic circuits and it can be applied on both pre-synthesized and post-synthesized gate-level circuits.

6.3 Debugging Multiple Errors

Table 7 presents results for remainder-generation, remainder partitioning, test generation, bug localization and debugging methods using multipliers and adders with multiple independent bugs. The first column (“Type”) indicates the types of benchmarks. The second (“Size”) and third columns (“#Bugs”) show the size of operands and number of bugs in each design, respectively. The fourth column (“RC(gates)”) shows th CPU time to generate the remainder. The fifth column (“RP (s)”) represents the required time for remainder partitioning, and the sixth column (“TG (s)”) represents the time of our test generation method. The seventh column (“Bug Loc. (s)”) shows the CPU time for bug localization algorithm. The eighth column (“DC (s)”) shows the debugging time to detect and correct all bugs. The next column (“total (RP+TG+BL+DC)(s)”) provides CPU time of our proposed approach which is the sum of remainder partitioning (RP), test generation (TG), bug localization (BL) and bug correction algorithm (DC) times. The tenth column (“#21(s)”) shows the required time of method presented in [21] using our implementation in Java. The next column (“Improvement”) shows the improvement provided by our debugging framework. Clearly, our approach is an order-of-magnitude faster than the most closely related approach [21], especially for larger multipliers as bug localization has an important effect. However, our performance is comparable with [21] for debugging adders since the number of
gates is small and the number of inputs is large and test generation time may surpass the speed up of our debugging method. The last column shows the required memory for our debugging approach.

Table 8 presents results for remainder-partitioning, test generation, bug localization and debugging methods using multipliers andadders with two dependent bugs. The first column ("Type") indicates the types of benchmarks. The second column ("Size") shows the size of operands. The third column ("RG(s)") shows the CPU time to generate the remainder. The fourth column ("RP(s)") represents the required time for remainder partitioning, and the fifth column ("TG(s)") represents the time of our test generation method. The sixth ("BL(s)") and sixth ("DC(s)") columns show the CPU time for bug localization and debugging time, respectively. Bug localization is relatively small in comparison with other scenarios since the intersection of faulty cones are not computed. The next column ("Total(s)") provides CPU time of our proposed approach which is the sum of remainder partitioning (RP), test generation (TG), bug localization (BL) and bug correction algorithm (DC) times. As the result shows, our approach can detect and correct multiple dependent bugs in reasonable time. We did not compare with any approaches since there are no existing approaches for detecting/fixing multiple dependent bugs. Finally, the last column shows the required memory for our debugging approach.

Table 9 shows our experimental results to debug one random bug. We have injected faults in different stages of the design: close to primary inputs 0−1/4, middle stages: 1/4−2/4 and 2/4−3/4, and close to primary outputs: 3/4−4/4. Column "RG(s)" shows the required time to perform functional rewriting (remainder generation time). Columns "RG", "TG", "BL", and "DC" present the required time for remainder generation, test generation, bug localization, and bug correction steps, respectively. The final column shows the overall debugging time. As it is shown in the result, incremental verification can remove the effect of bug location.

We have applied our method to Booth multipliers where partial products are generated using Booth architecture, compressor tree architecture is used for product accumulator, and final stage addition is done using Brent-kung architecture. Columns "RG", "TG", "BL", and "DC" present the required time for remainder generation, test generation, bug localization, and bug correction steps, respectively. The "total" column shows the overall debugging time. Finally, the last column shows the required memory for our debugging approach. The complexity of these benchmarks are comparable with array multipliers that are used in our previous experiments. Our method shows comparable performance for Booth multipliers as shown in Table 10. In fact, our method works efficiently on any combinational arithmetic circuits with different architectures since the performance of our approach is not dependent on exploiting similarity (e.g., half-adder structure) of pre-synthesized arithmetic circuits.

7 Conclusion and Future Work
In this paper, we presented an automated methodology for debugging arithmetic circuits. Our methodology consists of efficient directed test generation, bug localization, and bug correction algorithms. We used the remainder produced by equivalence checking methods to generate directed tests that are guaranteed to activate the source of the bug when the bug is unknown. We used the generated tests to localize the source of the bug and find suspicious areas in the design. We also developed an efficient debugging algorithm that uses the remainder as well as suspicious areas to locate and correct the bug without any manual intervention. We extended the proposed approach to automatically fix multiple bugs. Our experimental results demonstrated the effectiveness of our approach to solve debugging problem for large and complex arithmetic circuits by improving debug performance by an order-of-magnitude compared to the state-of-the-art approaches.

In our future research, we plan to reduce the complexity of debugging n dependent bug in general scenario by pruning the faulty candidates and using efficient algorithms. To solve the automated debugging of more than two dependent bugs, we plan to use memoization and map the problem to several integer subset sum problems to reduce the complexity of the problem.

8 Acknowledgments
This work was partially supported by grants from National Science Foundation (CNS-1441667) and Cisco Systems.

References
TABLE 7
CPU time and memory results for debugging of arithmetic circuits for multiple independent bugs. TO = timeout after 7200 sec. Bugs are inserted in the middle stages of the design as well as close to primary inputs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>#Bugs</th>
<th>RG(s)</th>
<th>B(s)</th>
<th>TGU(s)</th>
<th>LOC(s)</th>
<th>DC(s)</th>
<th>total (RG+BF+BG+BL+DS)(s)</th>
<th><a href="s">21</a></th>
<th>Imp.</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x8</td>
<td>8</td>
<td>0.15</td>
<td>0.01</td>
<td>0.04</td>
<td>0.03</td>
<td>0.47</td>
<td>0.69</td>
<td>1.37</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>16x16</td>
<td>8</td>
<td>0.21</td>
<td>0.01</td>
<td>0.07</td>
<td>0.03</td>
<td>0.78</td>
<td>1.37</td>
<td>2.51</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>32x32</td>
<td>8</td>
<td>1.41</td>
<td>0.03</td>
<td>1.2</td>
<td>0.02</td>
<td>1.62</td>
<td>4.25</td>
<td>10.07</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>64x64</td>
<td>8</td>
<td>3.65</td>
<td>0.03</td>
<td>3.8</td>
<td>0.04</td>
<td>4.32</td>
<td>10.43</td>
<td>26.37</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>128x128</td>
<td>8</td>
<td>14.86</td>
<td>0.03</td>
<td>15.9</td>
<td>0.07</td>
<td>31.48</td>
<td>55.39</td>
<td>178.89</td>
<td>3.51</td>
<td>3.51</td>
<td></td>
</tr>
<tr>
<td>256x256</td>
<td>8</td>
<td>15.1</td>
<td>0.003</td>
<td>3.78</td>
<td>0.01</td>
<td>45.21</td>
<td>73.17</td>
<td>250.03</td>
<td>3.41</td>
<td>3.41</td>
<td></td>
</tr>
<tr>
<td>Geometric Mean</td>
<td>8</td>
<td>3.65</td>
<td>0.003</td>
<td>3.78</td>
<td>0.01</td>
<td>45.21</td>
<td>73.17</td>
<td>250.03</td>
<td>3.41</td>
<td>3.41</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 8
CPU time and memory results for debugging of arithmetic circuits with two dependent bugs. Bugs are inserted in the middle stages of the design as well as close to primary inputs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>#Bugs</th>
<th>RG(s)</th>
<th>B(s)</th>
<th>TGU(s)</th>
<th>LOC(s)</th>
<th>DC(s)</th>
<th>total (RG+BF+BG+BL+DS)(s)</th>
<th><a href="s">21</a></th>
<th>Imp.</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x8</td>
<td>8</td>
<td>0.22</td>
<td>0.01</td>
<td>0.5</td>
<td>0.03</td>
<td>0.46</td>
<td>1.27</td>
<td>2.67</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>16x16</td>
<td>8</td>
<td>1.48</td>
<td>0.02</td>
<td>1.3</td>
<td>0.05</td>
<td>2.53</td>
<td>6.88</td>
<td>10.05</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>32x32</td>
<td>8</td>
<td>1.53</td>
<td>0.02</td>
<td>1.95</td>
<td>0.05</td>
<td>3.84</td>
<td>10.05</td>
<td>15.38</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>64x64</td>
<td>8</td>
<td>3.97</td>
<td>0.03</td>
<td>2.58</td>
<td>0.03</td>
<td>8.78</td>
<td>16.59</td>
<td>16.49</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>128x128</td>
<td>8</td>
<td>14.99</td>
<td>0.03</td>
<td>13.5</td>
<td>0.03</td>
<td>25.21</td>
<td>20.12</td>
<td>17.54</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>Geometric Mean</td>
<td>8</td>
<td>1.48</td>
<td>0.02</td>
<td>1.3</td>
<td>0.05</td>
<td>2.53</td>
<td>6.88</td>
<td>10.05</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 9
CPU time results for the functional rewriting (remender generation) and debugging of faulty integer multipliers with one bug. One bug is inserted in different stages of the design, close to primary inputs (0 − 1/4), middle stages (1/4 − 3/4), and close to primary outputs 3/4 − 4/4.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>#Bugs</th>
<th>RG(s)</th>
<th>B(s)</th>
<th>TGU(s)</th>
<th>LOC(s)</th>
<th>DC(s)</th>
<th>total (RG+BF+BG+BL+DS)(s)</th>
<th><a href="s">21</a></th>
<th>Imp.</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>256</td>
<td>0.01</td>
<td>0.04</td>
<td>0.88</td>
<td>0.001</td>
<td>0.47</td>
<td>1.37</td>
<td>2.51</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>128x128</td>
<td>256</td>
<td>0.01</td>
<td>0.06</td>
<td>1.12</td>
<td>0.001</td>
<td>0.54</td>
<td>1.97</td>
<td>2.67</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>256x256</td>
<td>256</td>
<td>0.01</td>
<td>0.07</td>
<td>2.58</td>
<td>0.001</td>
<td>1.01</td>
<td>3.51</td>
<td>2.91</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>Geometric Mean</td>
<td>256</td>
<td>0.01</td>
<td>0.06</td>
<td>1.12</td>
<td>0.001</td>
<td>0.54</td>
<td>1.97</td>
<td>2.67</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 10
Results of debugging of Booth multipliers (BP-CT-BK). Independent bugs are inserted in the middle stages of the design as well as close to the primary inputs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>#Bugs</th>
<th>RG(s)</th>
<th>B(s)</th>
<th>TGU(s)</th>
<th>LOC(s)</th>
<th>DC(s)</th>
<th>total (RG+BF+BG+BL+DS)(s)</th>
<th><a href="s">21</a></th>
<th>Imp.</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>8</td>
<td>0.21</td>
<td>0.01</td>
<td>0.81</td>
<td>0.01</td>
<td>0.91</td>
<td>1.23</td>
<td>2.51</td>
<td>1.91</td>
<td>1.91</td>
<td></td>
</tr>
<tr>
<td>128x128</td>
<td>8</td>
<td>0.42</td>
<td>0.01</td>
<td>2.51</td>
<td>0.01</td>
<td>2.86</td>
<td>3.75</td>
<td>3.75</td>
<td>1.91</td>
<td>1.91</td>
<td></td>
</tr>
<tr>
<td>256x256</td>
<td>8</td>
<td>0.42</td>
<td>0.01</td>
<td>2.51</td>
<td>0.01</td>
<td>2.86</td>
<td>3.75</td>
<td>3.75</td>
<td>1.91</td>
<td>1.91</td>
<td></td>
</tr>
<tr>
<td>64x64</td>
<td>8</td>
<td>0.21</td>
<td>0.01</td>
<td>0.81</td>
<td>0.01</td>
<td>0.91</td>
<td>1.23</td>
<td>2.51</td>
<td>1.91</td>
<td>1.91</td>
<td></td>
</tr>
<tr>
<td>128x128</td>
<td>8</td>
<td>0.42</td>
<td>0.01</td>
<td>2.51</td>
<td>0.01</td>
<td>2.86</td>
<td>3.75</td>
<td>3.75</td>
<td>1.91</td>
<td>1.91</td>
<td></td>
</tr>
<tr>
<td>256x256</td>
<td>8</td>
<td>0.42</td>
<td>0.01</td>
<td>2.51</td>
<td>0.01</td>
<td>2.86</td>
<td>3.75</td>
<td>3.75</td>
<td>1.91</td>
<td>1.91</td>
<td></td>
</tr>
<tr>
<td>Geometric Mean</td>
<td>8</td>
<td>0.21</td>
<td>0.01</td>
<td>0.81</td>
<td>0.01</td>
<td>0.91</td>
<td>1.23</td>
<td>2.51</td>
<td>1.91</td>
<td>1.91</td>
<td></td>
</tr>
</tbody>
</table>


