Automated Test Generation for Debugging Multiple Unknown Bugs in Arithmetic Circuits

Journal:	*Transactions on Computers*
Manuscript ID	TC-2017-04-0222
Manuscript Type:	Regular
Keywords:	Equivalence Checking, Directed Test Generation, Remainder-based Debugging, Bug Localization, Error Correction and Detection, Multiple Bugs Correction., Debugging based on Symbolic Algebra

Dear Editor-in-Chief,

This is an extended version of the paper that has appeared in the Proceedings of ACM/IEEE Design, Automation & Test in Europe (DATE 2016). The DATE paper presented some initial results on automated debugging of arithmetic circuits in the presence of **only one bug** (gate misplacement). This article has significant additional material, in particular, the following major contributions are new:

- Section 5 is added to introduce an approach to debug **multiple bugs**. Algorithm 4 is added to partition the remainder into sub-remainders, which are necessary to extend test generation, bug localization, and bug detection algorithms to work on multiple bugs. This section also includes examples showing how sub-remainders are utilized for generating directed tests, bug localization and bug correction.
- Section 5.1 is added to enable debugging of multiple independent. This section includes new procedure for detecting and correcting multiple independent bugs using illustrative examples.
- Section 5.2 is added to enable fixing of multiple dependent bugs (bugs with overlapping input cones). This section includes Algorithm 5 for correcting two dependent bugs as well as illustrative examples.
- Section 6.3 is added to show experimental results of detecting and correcting multiple bugs (both independent and dependent bugs). The results include the required time for remainder partitioning, test generation, bug localization, and bug correction algorithms.
- We have also included additional material in the introduction, background, related works, and provided detailed analysis of experimental results.

I believe these changes represent more than 40% additional material compared to the original DATE paper. Please let me know if you have any questions.

Best regards,

Farimah Farahmandi, Ph.D. student

University of Florida

Automated Test Generation for Debugging Multiple Unknown 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 equivalence checking 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 location 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 reminder non-zero. The generated tests are guaranteed to activate unknown bugs. We also propose a bug detection and correction 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 dependent as well as independent 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—Equivalence Checking, Directed Test Generation, Remainder-based Debugging, Bug Localization, Error Correction and Detection, 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. 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. Recent equivalence checking 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 detect 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 [1], [2] 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 unknown 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 [3], [4], [5] using a set of polynomials F. The specification of an arithmetic circuit can be modeled as a polynomial f_{spec} using a decimal 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 equivalence checking 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 decimal 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 detect and correct the bugs in the implementation. Fig. 1 shows an overview of

• *F. Farahmandi and P. Mishra are with the Department of Computer and Information Science and Engineering, University of Florida, FL, 32611. E-mail: {ffarahmandi,prabhat}@ufl.edu*

Manuscript received October 27, 2016

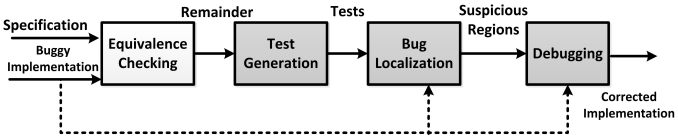

Fig. 1. Overview of our automated debugging framework. It consists of three important steps: test generation, bug localization, and automated debugging of arithmetic circuits.

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 an automated debugging to detect and correct the source of error. We have applied our method on large combinational arithmetic circuits to demonstrate the usefulness of our approach.

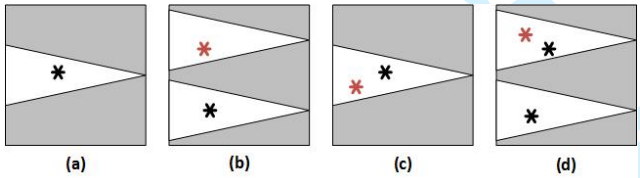

Fig. 2. Relative bugs' locations and their corresponding input cones of influence.

Figure 2 shows different scenarios for 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 present algorithms to detect and correct multiple independent bugs. In many cases, bugs may share input cones as shown in Figure 2 (c). In this paper, we also propose an algorithm to detect and correct multiple dependent unknown 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 detect and correct multiple bugs for various scenarios depicted in Figure 2. Existence of a non-zero remainder as a result of applying equivalence checking 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 detection and 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 remainder can

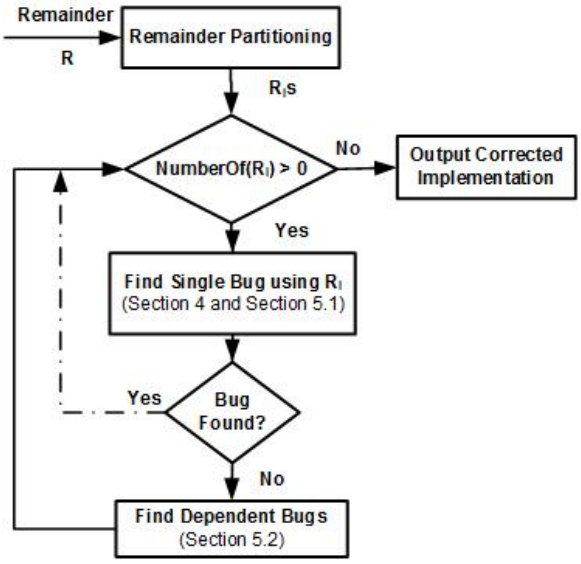

Fig. 3. Overview of different steps of our proposed debugging framework. Independent bugs are detected and corrected using the first looop with dotted line as described in Section 5.1. Debugging of dependent bugs are discussed in Section 5.2.

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 detect and correct 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 the algorithm of Section 5.2 to find dependent bugs responsible for sub-remainder R_i. The procedure will be repeated for all of the sub-remainders. To the best of our knowledge, our proposed method is the first attempt to automatically detect and 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 equivalence checking 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 correct 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 [6] and directed [2]. 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. Moreover, constraint generation is not possible when we do not have knowledge about the potential bug. A directed test generator, on the other hand, generates one test to target a specific functional scenario [2], [7]. Clearly, less effort is needed to reach the same coverage goal using directed tests compared to random or constrained-random tests. However, existing directed test generation methods require a fault list or desired functional behaviors that need to be activated [7]. These approaches cannot generate directed tests when the bug (faulty scenario) is unknown.

When effective tests are available, the source of error has to be localized. Most of the traditional debugging tools are based on techniques such as simulation, binary decision diagrams (like BDDs,*BMD [8]) and SAT solvers [9], [10]. Solving SAT problem results in finding suspicious bug locations. A SAT branching schema [10] is introduced to use reverse domination approach and reduce SAT solvers' efforts. However, all of these approaches suffer from state space explosion while dealing with large and complex arithmetic circuits. Furthermore, most of these approaches cannot provide concrete suggestions to fix bugs. Satisfiability modulo theory (SMT) solvers have been utilized to debug RTL designs [11]. Word-level MUXes are added to error candidate signals and the resultant formula is solved by a word-level SAT solver; however, these methods are dependent on existence of bug traces. The presented method of [12] suggests an error searching algorithm to reduce the potential error set. It uses masking error situations to find debugging priorities. This method relies on the coverage of tests generated by simulation. In [13], dynamic slicing for bug localization has been introduced; however, it requires subsequent error detection and correction. Several error correction efforts have been proposed over combinational circuits. In this paper, we propose an efficient framework to diagnose arithmetic circuit in order to detect and correct gate misplacement error. Existing method [15] is the most recent work in debugging arithmetic circuits that requires two rounds of verification process: backward rewriting and forward rewriting. As a result, [15] is slow and it is not scalable. Moreover, the approach cannot detect dependent bugs since the authors did not consider the effect of dependent bugs on each other.

The existing approaches either require manual intervention or not scalable. We propose an efficient, scalable and fully automated test generation, bug localization and debugging framework for arithmetic circuits.

3 Background: Remainder Generation

Several equivalence checking 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 [5], [16]. Another class of techniques are based on functional rewriting [4]. These methods can be applied on combinational [3] and sequential [17] Galois Filed $\mathbb{F}_{2^k}$ arithmetic circuits using Gröbner Basis theory [18] as well as signed/unsigned integer $\mathbb{Z}_{2^n}$ arithmetic circuits [4], [5], [19], [20].

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

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 $\mathbb{Z}_2$ ($x \in \mathbb{Z}_2 \rightarrow x^2 = x$). Variables can be selected from primary inputs/outputs as well as internal signals in the implementation. These polynomials are driven 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.

$$\begin{aligned} &z_1 = NOT(a) \rightarrow z_1 = 1 - a, \\ &z_2 = AND(a,b) \rightarrow z_2 = a.b, \\ &z_3 = OR(a,b) \rightarrow z_3 = a + b - a.b, \\ &z_4 = XOR(a,b) \rightarrow z_4 = a + b - 2.a.b \end{aligned} \tag{1}$$

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 have the lowest). By considering the derived variable ordering, each non-primary input variable which exists in the f_{spec} is replaced with its equivalent expression based on its polynomial. Then, the f_{spec_i} is updated and the process is continued on the updated $f_{spec_{i+1}}$ until we reach a zero polynomial or a polynomial that only contains primary inputs (remainder). Note that, using a fix variable (term) ordering to substitute the terms in the f_{spec_i}s, results in having a unique remainder [18]. 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_{spec}. The verification process starts from f_{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_{spec} is replaced with expression $(R+O-2.R.O)$. 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. ■

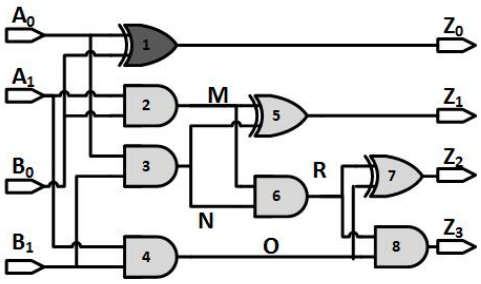

Fig. 4. Faulty gate-level netlist of a 2-bit multiplier

$$
\begin{aligned}
&f_{spec} : 8.Z_3 + 4.Z_2 + 2.Z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&step_1 : 4.R + 4.O + 2.z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&step_2 : 4.O + 2.M + 2.N + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&step_3(remiander) : 1.A_0 + 1.B_0 - 3.A_0.B_0
\end{aligned} \tag{2}
$$

4 Automated Debugging using Remainders

Our framework uses the remainder that is generated by equivalence checking 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 the remainder is zero, the implementation is bug-free [19]. Thus, when we have a non-zero polynomial as a remainder, any assignment to its variables that makes the decimal value of the remainder non-zero is a bug trace. In the proposed approach, we make use of the remainder to generate test cases to activate unknown faults. The test is guaranteed to activate the bug in the design. 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 possible the assignments to its variables such that it makes the decimal 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 ($i \neq 0 \in \mathbb{Z}, check(R = i)$). 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 decimal values of the remainder. Algorithm 1 shows the details of our test generation algorithm. The algorithm takes remainder (R) polynomial and primary inputs (PI) in the remainder as inputs, feeds binary values to PIs (s_i), and computes the total value of a term (T_j). The value of T_j is either one or zero as it is multiplication of some binary variables (line 4-5). The whole term value may be zero or equal to the term coefficient (C_{T_j}). Then, it computes the sum of the values of all the terms in the remainder to find the corresponding value of the remainder polynomial. If the summation (value) of all the terms is non-zero, the corresponding primary input assignments are added to the set of Tests (lines 8-9). The test generation algorithm can be easily 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 𝕋
4:      for different assignments s_i of PIs in R do
5:          for each term T_j ∈ R do
6:              if (T_j(s_i)) then
7:                  Sum+ = C_{T_j}
8:          if ( Sum != 0 ) then
9:              𝕋 = 𝕋 ∪ s_i
10: return 𝕋
```

Example 2: 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 decimal 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. The corresponding directed tests are shown in Table 1. ■

TABLE 1
Directed tests to activate fault shown in Fig. 4

A_1	A_0	B_1	B_0
X	1	X	0
X	0	X	1
X	1	X	1

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 simulate the tests 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 $E = \{e_1, e_2, .., e_n\}$. 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 that its 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 of the process, one fanout-gate is chosen from list L_{fo} and gate level netlist is traced backward until the gate g_i is reached where its inputs come from one of the fanouts in 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 $C_{e_i} = \{c_1, c_2, ..., c_j\}$ 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}s 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 ∈ E do
5:         find cones that construct e_i and put in C_ei
6:     C_S = C_e0
7:     for e_i ∈ E do
8:         C_S = C_S ∩ C_ei
9: return C_S
```

When the effect of the bug can be observed in multiple outputs, it means that the bug location is likely closer to the primary inputs as the error has been propagated through all the circuit. Thus, 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 of the circuit. We describe the details of debugging in Section 4.3.

Example 3: Consider the faulty 2-bit multiplier shown in Fig. 5. 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 simulated 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 $C_{Z_3} = \{2, 3, 4, 6, 8\}$. 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. ■

4.3 Error Detection and 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

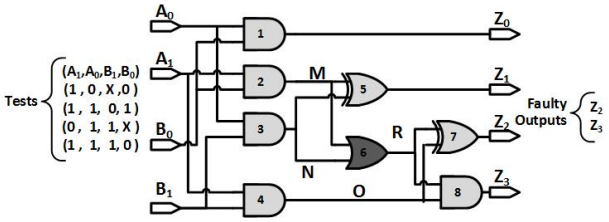

Fig. 5. Faulty gate-level netlist of a 2-bit multiplier with associated tests

TABLE 2
Remainder patterns caused by gate misplacement error

Suspicious Gate	Appeared Remainder's Pattern	Solution
AND (a,b)	P_1 : -a-b+2.a.b	S_1 : OR (a,b)
	P_2 : -a-b+3.a.b	S_2 : XOR (a,b)
OR (a,b)	P_1 : a+b-2.a.b	S_1 : AND (a,b)
	P_2 : a.b	S_2 : XOR (a,b)
XOR (a,b)	P_1 : a+b-3.a.b	S_1 : AND (a,b)
	P_2 :-a.b	S_2 : OR (a,b)

process in equivalence checking 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 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 2 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 $\mathbb{G}$. Next, the suspicious gates from the first level of $\mathbb{G}$ are considered and the remainder is scanned to check whether one of the patterns in Table 2 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 $\mathbb{G}$. Then, we repeat the process over the remaining gates in $\mathbb{G}$ until we find the source of the error.

Example 4: Consider the faulty circuit shown in Example 3. 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 2. 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 [15]). Now, we consider the gates in the second level. This time R^* matches with one of the patterns shown in Table 2. Based on Table 2, 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 $\mathbb{G}$. 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 $\mathbb{G}$ (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 2 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 2 (lines 13-16). Otherwise, g_i is correct and it will be removed from set $\mathbb{G}$ 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). As we want 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 by considering known weight of primary inputs and primary outputs, and moving from backward and forward considering the fact that 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 gate in the design is computed as the summation of inputs' weights (since they are mostly used in half adders [15]). In adders, the output's weight of all AND gates is computed as summation of input's weights. This process continues until the bug is detected or set $\mathbb{G}$ 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.

Example 5: We want to apply Algorithm 3 on the case shown in Example 4. 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 dictionary will be updated as $(M = 2.A_1.B_0, N = 2.A_0.B_1)$ at the end of this iteration. Now, the gate 6 is considered and the P_is 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 2 it will be fixed by replacing with an AND gate. ■

Algorithm 3 Error Detection/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 ∈ 𝔾 from level j do
7:              (a, b) = inputs(g_i)
8:              if !( each of (a, b) are from PI) then
9:                  a = dic.get(a)
10:                 b = dic.get(b)
11:             P_1 = ComputeP_1(a, b)
12:             P_2 = ComputeP_2(a, b)
13:             if (P_1 is found in R) then
14:              return gate g_i and solution S_1 from Table 2
15:             else if (P_2 is found in R) then
16:              return gate g_i and solution S_2 from Table 2
17:             else
18:                 remove g_i from 𝔾
19:                 dic.add(output(g_i), Expression(g_i(a, b)))
```

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 equivalence checking of an arithmetic circuit results to 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 misplaced gates.

If there are 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 equivalence checking procedure as well as the remainder generation. In other words, the remainder shows the effect of all (unknown) bugs in the implementation. Example 6 shows how the remainder is generated when there are two bugs in the implementation.

Example 6: In the circuit shown in Figure 6, 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 equivalence checking (remainder polynomial) can be computed as shown in Equation 3. ■

$$f_{spec} : 8.Z_3 + 4.Z_2 + 2.Z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0$$
$$step_1 : 4.R + 4.O + 2.z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0$$
$$step_2 : 4.O + 2.M + 2.N + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0$$
$$step_3(remiander) : R = A_0 + B_0 - 3.A_0.B_0 + 4.A_1 + 4.B_1 - 8.A_1.B_1 \quad (3)$$

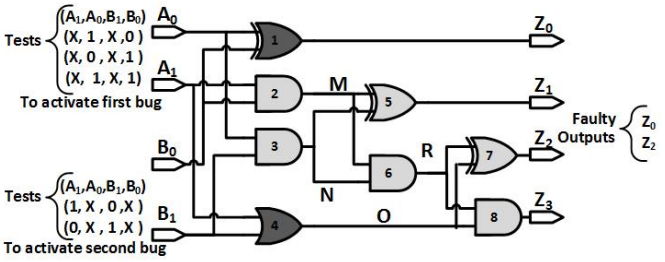

Fig. 6. Gate-level netlist of a 2-bit multiplier with two bugs (dark gates) as well as associated tests to activate them.

Detailed observation in the remainder generation procedure shows that the overall remainder can be considered as a summation of different individual bug's effect in the equivalence checking process. For instance, one part of the remainder shown in Example 3, 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 equivalence checking 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 7: Directed test to activate the buggy implementation of Example 6 are shown in Figure 5. The assignments make the first part of the remainder non-zero $(A_0 + B_0 - 3.A_0.B_0)$, 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 generated tests are simulated 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, bug detection, and correction for multiple bugs correction in two different scenarios: i) bugs with independent input cones (independent bugs), ii) bugs which share some input cones (dependent bugs).

5.1 Error Correction for Multiple Independent Bugs

We call two bugs independent of each other if they have different input cones (fan-ins). Figure 6 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 a summation of each individual bug's remainder (summation 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-remainders. 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 Remainder Partitioning

```
1:  procedure REMAINDER–PARTITIONING
2:      Input: Remainder R
3:      Output: Sub-remainders ℝ
4:      Sort terms of R based on their size
5:      R0 = largestTerm(R)
6:      ℝ = {R0}
7:      for each term t ∈ R do
8:          for each sub-remainder Ri ⊂ ℝ do
9:              if (Ri contains some of the variable t) then
10:                 Ri = Ri + t
11:             else
12:                 new Rj = t
13:                 ℝ = ℝ ∪ Rj
        return ℝ
```

Algorithm 4 takes the overall remainder R as input and returns the partitioned sub-remainders R_is. 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 (lines 7-8). If some of the variables which exist in the t already exist in terms of sub-remainder R_i, 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 to 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 8 illustrates the remainder partitioning procedure.

Example 8: Consider the faulty multiplier design shown in Figure 6 and corresponding remainder shown in Equation 3. 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 corresponding to tests of each sub-remainder are shown in Figure 6.

The generated tests are simulated 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 9.

Example 9: Having the directed tests shown in Figure 6, faulty outputs Z_0 and Z_2 as well as two sub-remainders computed in Example 8, 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. Therefore, 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 $P_2 = -1.A_0.B_0$ (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 10: Consider the faulty implementation of a 2-bit multiplier with two bugs as shown in Figure 7. Assume that gates 6 and 7 are misplaced with OR gates. It can be observed from Figure 7 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 equivalence checking on the circuit shown in Figure 7 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 misplaced 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 misplaced with an OR gate (single fault), the remainder will be computed as: $R_2 = 8.0.A_1.B_1 - 8.0.A_0.A_1.B_0.B_1$. 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: $R'_2 = 8.0.A_1.B_1 + 8.0.A_0.B_1 + 8.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1 + 8.0.A_0.A_1.B_0.B_1$. Now, it can be seen that the $R = R_1 + R'_2$. 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 coefficientsresulting in the term cancellation. ■

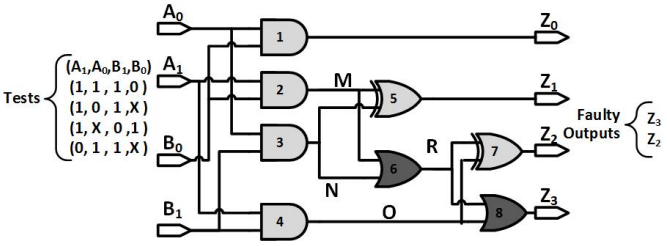

Fig. 7. Gate-level netlist of a 2-bit multiplier with two bugs (dark gates) which shares some input cones as well as associated tests to activate them.

As it can be observed from Example 10, 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 unknown dependent bugs is to use of Algorithm 1 in order to generate directed tests to activate unknown bugs. In the next step, tests are simulated 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 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, solution can be either OR gate or XOR gate based on Table 2) and we store them in set $\mathbb{R}$. To be able to detect the bugs, we are looking for n sub-remainders $R_i \in \mathbb{R}$ where their summation construct the original remainder R. This problem maps to the 0/1 knapsack problem. The complexity of the problem can be reduced to $O(n^2)$ using memoization. We show that how we can solve this problem for 2 multiple bugs as following. Note that, for two bugs we can reduce the complexity to $O(n)$ using efficient data structures. We illustrate this approach using two dependent bug for the rest of this Section.

To detect two dependent bugs, we are looking for two sub-remainders that their summation 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 10 , and also affected by faulty gate 6). Algorithm 5 is used to detect and correct two dependent bugs by finding two sub-remainders R_1 and R_2 where their summation is equal to R ($R = R_1 + R_2$). Therefore, it tries to find two equal polynomials: $R - R1$ and $R2$. 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 contains two major steps: first, it constructs two patterns for each potentially faulty gates based on Table 2 regarding the functionality of their input gates (lines 7-10). For each pattern, it computes the $R - P_i$ and it stores the result in a dictionary (lines 11-12). In the next step, each of the patterns P_j is checked to see if it exists in the dictionary (line 15). If such pair exists in the remainder, $R_1 = P_i$ and $R_2 = P_j$ are two sub-remainders that we are looking for and their corresponding gates are faulty gates and their solution can be found based on Table 2 (lines 16-18). Note that, by using hash map $\mathbb{R}$ the complexity of the algorithm is proportional to 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: Suspecious gates $\mathbb{G}$, remainder R
3: Output: Faulty gates and their solution
4: $\mathbb{P} = \{\}$ ▷ keeps patterns for all gates as well as corresponding solution of each pattern
5: $\mathbb{R} = \{\}$ ▷ keeps remainder minus all patterns
6: **for** each gate $g \in \mathbb{G}$ **do**
7: (a, b)=getInputPolynomials(g)
8: $P_1 = computeP1(a, b)$
9: $P_2 = computeP2(a, b)$
10: $\mathbb{P} = \mathbb{P} \cup \{P_1, P_2\}$
11: $\mathbb{R}.put((R - P_1), P_1)$
12: $\mathbb{R}.put((R - P_2), P_2)$
13: **for** each $P_j \in \mathbb{P}$ **do**
14: **if** P_j exists in $\mathbb{R}$ **then**
15: $P_i = \mathbb{R}.get(P_j)$
16: gate $g_i = \mathbb{P}.get(P_i)$ is faulty and get solution S_i from Table 2
17: gate $g_j = \mathbb{P}.get(P_j)$ is faulty and get solution S_j from Table 2

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). In arithmetic circuit implementations, most of the gates are connected to half-adders or they are in the last level of the design. Therefore, if we consider them as potentially faulty gates, their constructed patterns are equal to the exact remainder. However, if they are not in the last level of the design and they are not connected to a half-adder, the exact sub-remainder is also dependent on the structure of the next level. To illustrate the point, suppose that we assume that gate g_1 with functionality f_{g_1} may be misplaced by polynomial $f_{g_1'}$. Then the constructed pattern is computed as: $\Delta = f_{g_1'} - f_{g_1}$. If gate g_1 is connected to a half-adder with inputs g_1 and g_2, the exact sub-remainder is computed as: $\Delta - 2.\Delta.f_{g_2} + 2.\Delta.f_{g_2} = \Delta$. However, if it is only connected to an XOR gate g_2, the exact sub-remainder would be equal to: $\Delta - 2.\Delta.f_{g_2}$. Note that, if we have two cascaded bugs, the mentioned effect effect only may 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.

Example 11: Consider the faulty implementation of a 2-bit multiplier shown in Figure 7 with remainder: $R = 8.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$. Corresponding directed tests to activate existing bugs and faulty gates are shown in Figure 7. Potential faulty gates are computed based on faulty outputs Z_2 and Z_3 as gates $\{2, 3, 4, 6, 7, 8\}$. Patterns, their possible replacement as well remainder minus patterns are listed in Table 3. Note that, Table 3 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, the fourth column contains the highlighted polynomial of the column, and the highlighted polynomials are equal. It means that gate 6 and 7 are faulty and they should be substituted with AND gates. ■

6 EXPERIMENTS

6.1 Experimental Setup

The directed test generation, bug localization, and bug detection 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- [4] 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 gate misplacement 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 [4]. 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 [1] as well as random test generation. To start our debugging procedure, we use the generated counterexamples in test generation phase and find faulty primary outputs. Then, we run the bug localization algorithm that takes faulty outputs as input. In the next step, we apply our debugging algorithm on suspicious areas that bug localizer has identified. Note that, our debugging procedure does not need the suspicious gates to work and in the worst case, it considers all of the gates suspicious. However, using bug localization algorithm improves our method drastically. As remainder may explode

TABLE 3
Patterns for potential faulty gates Example 11

$Gate\#$	Pattern	solution	Remainder minus pattern
2	$2.A_1 + 2.B_0 - 4.A_1.B_0$	OR	$-2.A_1 - 2.B_0 + 8.A_1.B_1 + 16.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
	$2.A_1 + 2.B_0 - 6.A_1.B_0$	XOR	$-2.A_1 - 2.B_0 + 8.A_1.B_1 + 18.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
3	$2.A_0 + 2.B_1 - 4.A_0.B_1$	OR	$-2.A_0 - 2.B_1 + 8.A_1.B_1 + 12.A_1.B_0 + 16.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
	$2.A_0 + 2.B_1 - 6.A_0.B_1$	XOR	$-2.A_0 - 2.B_1 + 8.A_1.B_1 + 12.A_1.B_0 + 18.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
4	$4.A_1 + 4.B_1 - 8.A_1.B_1$	OR	$-4.A_1 - 4.B_1 + 16.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
	$4.A_1 + 4.B_1 - 12.A_1.B_1$	XOR	$-4.A_1 - 4.B_1 + 20.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
6	$4.A_0.B_1 + 4.A_1.B_0 - 8.A_0.A_1.B_0.B_1$	AND	$8.A_1.B_1 + 8.A_1.B_0 + 8.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1 + 8.A_0.A_1.B_0.B_1$
	$4.A_0.A_1.B_0.B_1$	XOR	$8.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1 + 4.A_0.A_1.B_0.B_1$
7	$4.A_0.B_1 + 4.A_1.B_0 + 4.A_1.B_1 - 8.A_0.A_1.B_0 - 8.A_1.B_0.B_1 + 4.A_0.A_1.B_0.B_1$	AND	$4.A_1.B_1 + 8.A_1.B_0 + 8.A_0.B_1 - 8.A_0.A_1.B_0 - 8.A_1.B_0.B_1 + 8.A_0.A_1.B_0.B_1$
	$4.A_0.A_1.B_0 + 4.A_1.B_0.B_1 - 4.A_0.A_1.B_0.B_1$	OR	$8.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 20.A_0.A_1.B_0 - 20.A_1.B_0.B_1 + 4.A_0.A_1.B_0.B_1$
8	$8.0.A_1.B_1 + 8.0.A_0.B_1 + 8.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1 + 8.0.A_0.A_1.B_0.B_1$	AND	$8.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
	$8.A_0.A_1.B_0 + 8.A_1.B_0.B_1 - 8.A_0.A_1.B_0.B_1$	XOR	$8.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 24.A_0.A_1.B_0 - 24.A_1.B_0.B_1 + 8.A_0.A_1.B_0.B_1$

when bugs are closer to the primary outputs, these bugs are harder to detect especially when the size of the circuit is very large. On the other hand, the bugs that are closer to the primary inputs are easier to detect. 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 [15]. We use the benchmarks obtained from the authors [15]. However, we have implemented their algorithm to be able to compare our method with their method. To enable fair comparison, similar to [15], we randomly inserted bugs (gate changes) in the middle stages of the circuits. We improved the run-time complexity of presented method in [21] by using efficient data structures such as hash maps and sorted sets.

6.2 Debugging a Single Error

Table 4 presents results for test generation, bug localization and debugging methods using multipliers and adders. The first column indicates the types of benchmarks. The second and third 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 indicates results for directed test generation method presented in [1] by using SMV model checker [22] (We give the model checker the advantage of knowing the bug). The fifth column represents results of random test generation method (time to generate the first counterexample using the random technique). The sixth column represents the time of our test generation method that generates multiple tests. As it can be observed from Table 4, our method has improved directed test generation time by several orders of magnitude. The seventh column shows the CPU time for bug localization algorithm. The eighth column shows the debugging time of [15] using our implementation in Java. The next column provides CPU time of our proposed approach which is the summation of test generation (TG), bug localization (BL) and debugging/correction (DC) time. The last column shows the improvement provided by our debugging framework. Clearly, our approach is an order-of-magnitude faster than the most closely related approach [15], 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 in test generation of the first row (post-synthesized multiplier with 4-bit operands) of Table 4, the reported results are the average of the nine possible scenarios shown in Table 5.

Table 5 presents the debugging results of 4-bit post-synthesized multiplier. The first column shows a possible set of gate misplacement faults. Time to generate the first counterexample using [1] and random techniques are reported in second and third columns, respectively. The fourth column shows the number of directed tests generated by our approach to activate the bug (each of them activates the bug). The fifth column lists the outputs that are affected by the fault (activated by the respective tests reported in the fourth column). The sixth column shows the number of random tests required to cover all of our directed tests. It demonstrates that even for such small circuits, using random tests to activate the error is impractical. The last column shows our test generation time. As mentioned earlier, the average of these scenarios is reported in the first row of Table 5.

The experimental results demonstrated 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 misplacement 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 6 presents results for remainder-partitioning, test generation, bug localization and debugging methods using multipliers and adders with multiple independent bugs. The first column indicates the types of benchmarks. The second and third columns show the size of operands and number of bugs in each design, respectively. The fourth column represents the required time for remainder partitioning, and the fifth column represents the time of our test generation method. The sixth column shows the CPU time for bug localization algorithm. The seventh column shows the debugging time to detect and correct all bugs. The next column provides CPU time of our proposed approach which is the summation of remainder partitioning (RP), test generation (TG), bug localization (BL) and bug detection algorithm (DC) times. The ninth column shows the required time of method presented in [15] using our implementation in Java. The last column shows the improvement provided by our debugging framework. Clearly, our approach is an order-of-magnitude faster than the most closely related approach [15], especially for larger multipliers as bug localization has an important effect. However, our performance is comparable with [15] for debugging adders since the

TABLE 4
Debugging results of Arithmetic Circuits . TO = timeout after 3600 sec; MO = memory out of 8 GB

Benchmark			Test Generation (TG)			Bug Localization (BL)	Debugging/Correction (DC)		
Type	Size	# Gates	[1] (s)	Random(s)	Our TG(s)	Bug Loc.(s)	[15]	Our (TG+BL+DC)	Improvement
post-syn. Multipliers	4	72	1.88	0.02	0.01	0.001	0.2	0.02	10x
	16	1632	42.69	1.48	0.32	0.03	4.32	0.78	5.5x
	32	6848	205.66	3.03	0.82	0.16	18.50	2.40	10.94x
	64	28K	MO	16.97	1.65	0.83	151.05	13.63	11.08x
	128	132K	MO	66.52	3.83	5.1	1796.50	52.91	33.95x
	256	640K	MO	TO	15.65	22.39	TO	205.01	-
pre-syn. Multipliers	4	94	1.27	0.04	0.01	0.001	0.17	0.03	5.6x
	16	1860	43.11	1.93	0.4	0.03	4.45	0.83	5.36x
	32	7812	189.50	5.69	0.87	0.2	23.1	2.67	8.65x
	64	32K	MO	29.07	1.77	0.8	180.3	14.91	12.09x
	128	129K	MO	83.60	4.1	3.8	1743.07	47.74	36.51x
	256	521K	MO	TO	12.44	15.83	TO	170.48	-
post-syn. Adder	64	573	154.97	1.51	0.5	0.01	3.12	0.71	4.39x
	128	1251	MO	3.48	1.07	0.05	6.60	1.69	3.90x
	256	2301	MO	10.64	3.09	0.05	17.32	4.27	3.35x
pre-syn. Adder	64	444	128.12	1.15	0.35	0.01	2.95	0.51	5.78x
	128	880	MO	4.40	0.84	0.03	6.46	1.12	5.76x
	256	1698	MO	9.10	2.23	0.1	16.18	3.54	2.05x

TABLE 5
Test Generation for 4-bit multiplier with 8 bits outputs # Gates = 72

Faults	[1]	Ran. tests(ms)	#tests	Faulty outputs	# Ran.	Our TG(s)
$XOR \rightarrow AND$	1.48	47.70	18	Z_7, Z_6, Z_5, Z_4	2632	0.01
$XOR \rightarrow OR$	2.12	25.95	4	Z_2	2945	0.01
$XOR \rightarrow AND$	1.95	19.21	128	Z_4	2292	0.01
$XOR \rightarrow OR$	2.27	26.43	12	Z_6, Z_5, Z_4, Z_3	2945	0.05
$XOR \rightarrow AND$	1.03	16.31	14	Z_6, Z_5, Z_4, Z_3, Z_2	2369	0.02
$AND \rightarrow XOR$	2.44	0.47	3	Z_6, Z_5, Z_4, Z_3, Z_2	1881	0.01
$AND \rightarrow OR$	2.20	1.90	2	Z_7, Z_6, Z_5	2258	0.01
$AND \rightarrow XOR$	0.89	44.17	148	Z_7, Z_6, Z_5, Z_4	2164	0.03
$OR \rightarrow AND$	2.52	11.51	148	Z_6	2920	0.01
Average	1.88	21.52	53	-	2489.55	0.01

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.

If remainder cannot be partitioned and Algorithm 3 cannot find a single source of error, we know that there are dependent bugs in the implementation. Therefore, Algorithm 5 is used to find dependent bugs. In this step, suspicious cones, as well as patterns, are generated for single bug detection phase will be reused. We consider all of the gates in the suspicious cones as potential faulty gates (instead of intersecting faulty cones) to make sure that all sources of errors can be found. Algorithm 5 uses hash maps that map the string versions of polynomials to the possible solutions to able to perform a faster lookup and achieve a linear computation complexity proportional with the number of suspicious gates.

Table 7 presents results for remainder-partitioning, test generation, bug localization and debugging methods using multipliers and adders with two dependent bugs. The first column indicates the types of benchmarks. The second column shows the size of operands. The third column represents the required time for remainder partitioning, and the fourth column represents the time of our test generation method. The fifth and sixth columns show the CPU time for bug localization and debugging time, respectively. Bug localization time is relatively small in comparison with other scenarios since the intersection of faulty cones are not computed. The next column provides CPU time of our proposed approach which is the summation of remainder partitioning (RP), test generation (TG), bug localization (BL) and bug detection 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.

7 Conclusion

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 detect and correct the bug without any manual intervention. We extended the proposed approach to automatically detect and correct 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.

References

[1] M. Chen and P. Mishra, "Functional test generation using efficient property clustering and learning techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 396–404, 2010.

[2] M. Chen, X. Qin, H. Koo, and P. Mishra, *System-level Validation - high-level modeling and directed test generation techniques*. Springer, 2012.

[3] J. Lv, P. Kalla and F. Enescu, "Efficient grbner basis reductions for formal verification of galois field multipliers," in *Design Automation and Test in Europe Conference(DATE)*, 2012, pp. 899–904.

[4] M. J. Ciesielski, C. Yu, W. Brown, D. Liu and A. Rossi, "Verification of gate-level arithmetic circuits by function extraction," in *IEEE/ACM International Conference on Computer Design Automation(DAC)*, 2015, pp. 1–6.

[5] F. Farahmandi and B. Alizadeh, "Grobner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," in *Microprocessor and Microsystems - Embedded Hardware Design*, 2015, pp. 83–96.

[6] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Vinov and a. Vinov, "Genesys-pro: Innovations in test program generation for functional processor verification," vol. 2, no. 38, Mar-Apr 2004, pp. 84–93.

TABLE 6
Debugging time of multipliers for multiple independent bugs. TO = timeout after 7200 sec.

Type	Size	#Bugs	RP(s)	TG(s)	Bug Loc.(s)	DC(s)	total (RP+TG+BL+DC)(s)	[15]	Improvement
post_syn. Multipliers	8x8	4	0.001	0.04	0.03	0.57	0.64	1.7	2.65x
		8	0.001	0.07	0.03	0.86	0.97	2.5	2.57x
	16x16	4	0.003	0.77	0.01	1.82	2.60	6.03	2.31x
		8	0.003	1.2	0.02	2.62	3.84	10.07	2.70x
	32x32	4	0.003	1.86	0.64	5.02	7.52	26.37	3.49 x
		8	0.003	2.08	1.18	9.4	12.67	43.98	3.47x
	64x64	4	0.006	5.65	3.9	38.48	48.03	178.89	3.72x
		8	0.006	7.06	4.7	65.31	78.07	250.07	3.206x
	128x128	4	0.008	11.59	10.1	124.52	146.22	1946.1	13.30x
		8	0.008	25.67	20.87	235.88	282.43	2337.56	8.28x
	256x256	4	0.012	39.58	70.65	508.42	618.66	TO	-
		8	0.012	65.21	122.01	906.22	1093.45	TO	-
pre_syn. Multipliers	8x8	4	0.001	0.44	0.03	0.35	0.82	1.73	2.11x
		8	0.001	0.5	0.03	0.66	1.19	2.67	2.24x
	16x16	4	0.002	1.3	0.05	2	3.35	7.4	2.21x
		8	0.002	1.90	0.05	2.87	4.79	10.05	2.1x
	32x32	4	0.003	2.08	0.73	5.8	8.61	30.34	3.52x
		8	0.003	3.23	1.31	11.98	16.52	43.18	2.61
	64x64	4	0.001	5.94	4.5	38.22	48.66	194	3.99x
		8	0.005	7.91	8.9	84.7	101.52	225.85	2.22x
	128x128	4	0.006	13.5	15.09	170.46	199.05	2036.37	10.36x
		8	0.006	22.48	26.72	207.88	257.09	2260.6	8.79x
	256x256	4	0.01	26.75	39.16	653	718.92	TO	-
		8	0.01	59.13	77.34	866.18	1002.66	TO	-
post_syn. Adders	64x64	4	0.004	1.09	0.07	0.37	1.53	3.43	2.24x
		8	0.003	2.63	0.12	0.47	3.23	3.85	1.19x
	128x128	4	0.005	3.34	0.2	0.71	4.25	7.59	1.78x
		8	0.01	5.41	0.37	1.25	7.24	8.72	1.2x
	256x256	4	0.01	8	0.3	5.62	13.93	19.87	1.42x
		8	0.01	13.44	0.8	9.94	24.19	25.93	1.07x
pre_syn. Adders	64x64	4	0.002	1.08	0.08	0.3	1.56	3.36	2.15x
		8	0.006	2.12	0.08	0.42	2.63	3.56	1.35x
	128x128	4	0.008	3.39	0.2	0.79	4.39	7.26	1.65x
		8	0.009	5.57	0.42	1.7	7.69	8.31	1.04x
	256x256	4	0.01	6.24	0.28	5.38	11.91	19.13	1.61x
		8	0.01	11.52	0.81	8.27	20.61	23.35	1.13x

TABLE 7
Debugging time of multipliers for two dependent bugs.

Type	Size	RP(s)	TG(s)	BL(s)	DC(s)	total (s)
post_syn. Mul.	8	0.001	0.1	0.01	0.98	1.09
	16	0.002	0.35	0.02	2.23	2.61
	32	0.002	0.96	0.08	13.92	14.94
	64	0.004	3.77	0.2	77.12	81.1
	128	0.008	8.06	0.6	241.05	249.71
	256	0.012	31.8	36.02	1099.96	1167.79
pre_syn. Mul.	8	0.001	0.1	0.01	0.91	1.02
	16	0.001	0.77	0.01	5	5.78
	32	0.002	1.03	0.08	13.54	14.65
	64	0.003	4.65	0.1	96.3	101.05
	128	0.005	7.88	0.6	220.22	228.70
	256	0.01	19.41	22.05	982.9	1024.37
post_syn. Mul.	64	0.001	01.18	0.01	0.55	1.74
	128	0.011	5.4	0.02	3.47	8.90
	256	0.011	16.09	0.1	9.42	25.62
pre_syn. Add.	64	0.003	1.13	0.01	0.53	1.67
	128	0.008	6.3	0.01	2.36	8.68
	256	0.01	10.97	0.08	15.04	24.10

[7] L. Liu and S. Vasudevan, "Efficient validation input generation in rtl by hybridized source code analysis," in *Design Automation and Test in Europe(DATE)*, 2011, pp. 1–6.

[8] R.E. Bryant and Y.A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *Proceedings of Design Automation Conference (DAC)*, 1995, pp. 535–541.

[9] H. Mangassarian, A. Veneris, S. Safapour, M. Benedetti and D. Smith, "A performance-driven qbf-based iterative logic array representation with applications to verification, debug and test," in *IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 240–245.

[10] B. Le, H. Mangassarian, B. Keng and A. Veneris, "Non-solution implications using reverse domination in a modern sat-based debugging enviroment," in *Design Automation and Test in Europe(DATE)*, 2012, pp. 629–634.

[11] S. Mirzaeian, F. Zheng and K. T. Chen, "Rtl error diagnosis using a word-level sat-solver," in *Proc. IEEE Int. Test Conference (ITC)*, 2008, pp. 1–8.

[12] K. Chang, I. Markov and V. Bertacco, "Accurate rank ordering of error candidates for efficient hdl design debugging," in *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2009, pp. 272–284.

[13] U. Repinski, H. Hantson, M. Jenihhin, J. Raik, R. Ubar, G. Di Guglielmo, G. Pravadelli and F. Fummi, "Combining dynamic slicing and mutation operators for esl correction," in *Proc. 17th IEEE Euro. Test Symp*, 2012, pp. 1–6.

[14] K. Chang, I. Markov and V. Bertacco, "Fixing design errors with counterexamples and resynthesis," in *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2007, pp. 944–949.

[15] S. Ghandali, C. Yu, W. Brown, and M. Ciesielski, "Logic debugging of arithmetic circuits," in *IEEE Computer Society Annual Symposium on VLSI(ISVLSI)*, 2015.

[16] N. Shekhar, P. Kalla and F. Enescu, "Equivalence verification of polynomial datapaths using ideal membership testing," in *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2006, pp. 1188–1201.

[17] X. Sun, P. Kalla, T. Pruss, and F. Enescu, "Formal verification of sequential galois field arithmetic circuits using algebraic geometry," in *Design Automation and Test in Europe(DATE)*, 2015, pp. 1623–1628.

[18] D. Cox, J. Little, and D. O'Shea, *Ideals, varieties, and algorithms*. Springer, 1997.

[19] O. Wienand, M. Welder, D. Stoffel, W. Kunz and G. M. Greuel, "An algebraic approach for proving data correctness in arithmetic data paths," in *Computer Aided Verification (CAV)*, 2008, pp. 473–486.

[20] F. Farahmandi, B. Alizadeh and Z. Navabi, "Effective combination

of algebraic techniques and decision diagrams to formally verify large arithmetic circuits," in *IEEE Computer Society Annual Symposium on VLSI(ISVLSI)*, 2014, pp. 338–343.

[21] F. Farahmandi and P. Mishra, "Automated test generation for debugging arithmetic circuits," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1351–1356.

[22] *The Cadence SMV Model Checker*, Cadence Berkeley Lab, Available at http://www.kenmcmil.com.

Farimah Farahmandi is pursuing Ph.D in Department of Computer and Information Science and Engineering (CISE) at the University of Florida. She received her B.S. and M.S. from department of Electrical and Computer Engineering (ECE), University of Tehran, Iran in 2010 and 2013 respectively. Her current research interests include formal verification and debugging, hardware security validation and post-silicon validation and debug. She was a research intern at Cisco advanced security research group.

Prabhat Mishra is a Professor in the Department of Computer and Information Science and Engineering at the University of Florida. His research interests include design automation of embedded systems, energy-aware computing, hardware security and trust, system validation and verification, reconfigurable architectures, and post-silicon debug. He received his Ph.D. in Computer Science and Engineering from the University of California, Irvine. He has published five books and more than 125 research articles in premier international journals and conferences. His research has been recognized by several awards including the NSF CAREER Award, IBM Faculty Award, three best paper awards, and EDAA Outstanding Dissertation Award. Prof. Mishra currently serves as the Deputy Editor-in-Chief of IET Computers & Digital Techniques, and as an Associate Editor of ACM Transactions on Design Automation of Electronic Systems, IEEE Transactions on VLSI Systems, and Journal of Electronic Testing. He has served on many conference organizing committees and technical program committees of premier ACM and IEEE conferences. He is currently serving as an ACM Distinguished Speaker. Prof. Mishra is an ACM Distinguished Scientist and a Senior Member of IEEE.

Automated Test Generation for Debugging Arithmetic Circuits

Farimah Farahmandi and Prabhat Mishra
Department of Computer and Information Science and Engineering
University of Florida, USA
{farimah,prabhat}@cise.ufl.edu

***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 equivalence checking 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 and can be infeasible for complex arithmetic circuits. In this paper, we present an automated test generation and bug localization technique for debugging arithmetic circuits. This paper makes two important contributions. We propose an automated approach for generating directed tests by suitable assignments of input variables to make the reminder non-zero. The generated tests are guaranteed to activate the unknown bug. We also propose a bug detection and correction technique by utilizing the patterns of remainder terms as well as the intersection of regions activated by the generated tests. Our experimental results demonstrate that the proposed approach can be used for automated debugging of complex arithmetic circuits.**

I. 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. 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. Optimized and custom arithmetic architectures are required to meet these 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. Hence, the 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 detect and fix it. We can always keep on generating random tests, in the hope of activating the bug; however, random test generation is neither scalable nor efficient when designs are large and complex. Existing directed test generation techniques [1], [2] are promising only when the list of faults (bugs) is available. However, they are not applicable when bugs are not known a priori. We propose a directed test generation technique that is guaranteed to activate unknown bugs (if any). The generated tests would also help for faster bug localization.

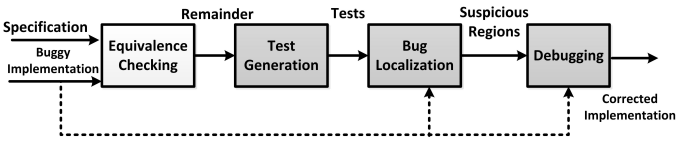

Fig. 1. Overview of our automated debugging framework. It consists of three important steps: test generation, bug localization, and automated debugging of arithmetic circuits.

Existing arithmetic circuits verification approaches have focused on checking the equivalence between the specification of a circuit and its implementation. Both specification and implementation are represented using polynomials. If the implementation is equivalent to the specification, the result of equivalence checking is a zero polynomial; otherwise, it produces a polynomial containing primary inputs as variables. We call this polynomial *remainder*. Any assignment to remainder's variables that makes the remainder to have a non-zero decimal value, generates one counterexample. Remainder generation is one time effort and multiple counterexamples (directed tests) can be generated from one remainder.

In this paper, we present a framework for directed test generation and automated debugging of datapath intensive applications using the remainder. Fig. 1 shows an overview of our proposed framework. Our method generates directed test vectors that are guaranteed to activate the bug (if any). The basic idea is to find input assignments such that the remainder polynomial becomes non-zero. 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 the bug. 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 an automated debugging to detect and correct the source of error. We have applied our method on large combinational arithmetic circuits to demonstrate the usefulness of our approach.

The remainder of the paper is organized as follows. We dis-

cuss related work in Section II. Section III gives an overview about equivalence checking and remainder generation. Section IV discusses our framework for directed test generation and bug localization. Section V presents our experimental results. Finally, Section VI concludes the paper.

II. 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 [3] and directed [2]. 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. Constraint generation is not possible when we do not have knowledge about the potential bug. A directed test generator, on the other hand, generates one test to target a specific functional scenario [2], [4]. However, existing directed test generation methods require a fault list or desired functional behaviors that need to be activated [4]. These approaches cannot generate directed tests when the bug (faulty scenario) is unknown.

When effective tests are available, the source of error has to be localized. Most of the traditional debugging tools are based on techniques such as simulation, binary decision diagrams (like BDDs,*BMD [5]) and SAT solvers [6], [7]. However, All of these approaches suffer from state space explosion usually accompanying the real designs such as large and complex arithmetic circuits. The presented method in [8] suggests an error searching algorithm to reduce the potential error set. Several error correction efforts have been proposed over combinational circuits. The existing approaches either require manual intervention or not scalable. The work of [17] presents an automated approach that scans the whole design in order to detect a bug but has scalability concerns. We propose an efficient test generation, bug localization and debugging framework that is fully automated and scalable.

III. Background: Remainder Generation

Several equivalence checking 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 [9], [10]. Another class of techniques are based on functional rewriting [11]. These methods can be applied on combinational [17] and sequential [12] Galois Filed $\mathbb{F}_{2^k}$ arithmetic circuits using Gröbner Basis theory [13] as well as signed/unsigned integer $\mathbb{Z}_{2^n}$ arithmetic circuits [18], [10], [14].

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

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 $\mathbb{Z}_2$ ($x \in \mathbb{Z}_2 \rightarrow x^2 = x$). Equation 1 shows the corresponding polynomial of *NOT, AND, OR, XOR* gates.

$$\begin{aligned} &z_1 = NOT(a) \rightarrow z_1 = 1 - a, \\ &z_2 = AND(a, b) \rightarrow z_2 = a.b, \\ &z_3 = OR(a, b) \rightarrow z_3 = a + b - a.b, \\ &z_4 = XOR(a, b) \rightarrow z_4 = a + b - 2.a.b \end{aligned} \tag{1}$$

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. Example 1 shows the verification process of a faulty 2-bit multiplier.

Example 1: We want to verify a 2-bit multiplier with gate-level netlist shown in Fig. 2. Suppose that, we deliberately insert a bug in the circuit shown in Fig. 2 by putting the OR gate with inputs (A_1, B_0) instead of an AND gate. The specification of a 2-bit multiplier is shown by f_{spec}. The verification process starts from f_{spec} and replaces its terms one by one using information derived from implementation polynomials as shown in Equation 2. For instance, term $4.Z_2$ from f_{spec} is replaced with expression $(R + O - 2.R.O)$. 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.

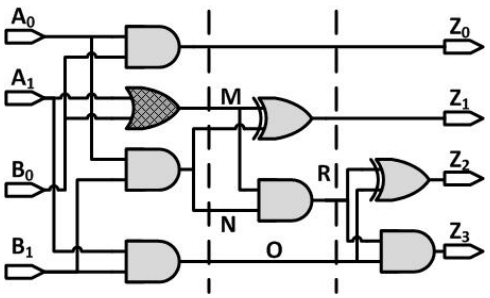

Fig. 2. Faulty gate-level netlist of a 2-bit multiplier

$$\begin{aligned} &f_{spec} : 8.Z_3 + 4.Z_2 + 2.Z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\ &step_1 : 4.R + 4.O + 2.z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\ &step_2 : 4.O + 2.M + 2.N + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\ &step_3(remiander) : 2.A_1 + 2.B_0 - 4.A_1.B_0 \end{aligned} \tag{2}$$

IV. Automated Debugging using Remainders

Our framework uses the remainder that is generated by equivalence checking in Section III. 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 existing bug which resides in the suspicious area. We describe each of these steps in detail in the following sections.

A. Directed Test Generation

It has been shown that if the remainder is zero, the implementation is bug-free [14]. Thus, when we have a non-zero polynomial as a remainder, any assignment to its variables that makes the decimal value of the remainder non-zero is a bug trace. 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 all of the assignments to its variables such that it makes the decimal 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 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 total value of the remainder polynomial ($i \neq 0 \in \mathbb{Z}, check(R = i)$). 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 parallel algorithm to find all possible assignments which produce non-zero decimal values of the remainder. Algorithm 1 shows the details of our test generation algorithm. The algorithms gets remainder R polynomial and primary inputs (PI) in the remainder as inputs and feeds binary values to PIs (s_i) and computes the total value of a term (T_j). If the summation (value) of all the terms is non-zero, the corresponding primary input assignments are added to the set of Tests (lines 8-9).

Algorithm 1 Directed Test Generation Algorithm

```
1: procedure TEST–GENERATION
2:     Input: Remainder, R
3:     Output: Directed Tests, Tests
4:     for different assignments s_i of PIs in R do
5:         for each term T_j ∈ R do
6:             if (T_j(s_i)) then
7:                 Sum+ = C_{T_j}
8:         if ( Sum != 0 ) then
9:             Tests = Tests ∪ s_i
       return Tests
```

Example 2: Consider the faulty circuit shown in Fig. 2 and the remainder polynomial $R = 2.(A_1 + B_0 - 2.A_1.B_0)$. The only assignments that make R to have a non-zero decimal value ($R = 2$) are $(A_1 = 1, B_0 = 0)$ and $(A_1 = 0, B_0 = 1)$. These are the only scenarios that make difference between functionality of an AND gate and an OR gate. Otherwise, the fault will be masked. Compact directed test are shown in Table I.

TABLE I
DIRECTED TESTS TO ACTIVATE FAULT SHOWN IN FIG. 2

A_1	A_0	B_1	B_0
1	X	X	0
0	X	X	1

B. 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 tests generated in the previous section. We simulate the tests and compare the outputs with the golden outputs and keep track of faulty outputs in set $E = \{e_1, e_2, .., e_n\}$. Each e_i denotes one of the erroneous outputs. To localize the bug, we partition the gate-level netlist such that fanout free cones (set of gates that are directly connected together) of the implementation are found.

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 ∈ 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 ∈ E do
8:         C_S = C_S ∩ C_{e_i}
       return C_S
```

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 sets of cones $C_{e_i} = \{c_1, c_2, ..., c_j\}$ 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}s 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).

If simulating all of the tests show the effect of the faulty behavior in just one of the outputs, we can conclude that the location of the bug is in the cone that generates this output. Otherwise, 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 of the circuit. We describe the details of debugging in Section IV-C.

Example 3: Consider the faulty 2-bit multiplier shown in Fig. 3. 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 IV-A. Tests are simulated 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 $C_{Z_3} = \{2, 3, 6, 4, 8\}$. 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 a source of the error.

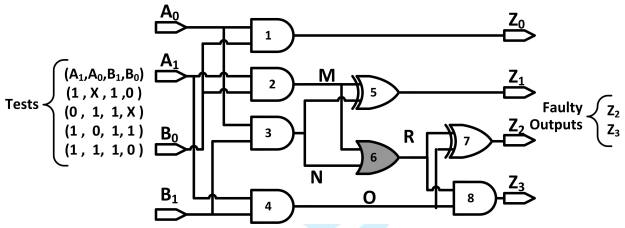

Fig. 3. Faulty gate-level netlist of a 2-bit multiplier with associated tests

C. Error Detection and 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 equivalence checking after this level, which prevents errors from propagating any further. In Example 1, the first level OR gate is placed by mistake instead of an AND gate. Let us consider the effect of the bug from algebraic point of view: the equivalent algebraic value of M is $M = A_1 + B_0 - A_1.B_0$ in the erroneous implementation; however, in the correct implementation, M should be equal to $M^* = A_1.B_0$. Thus, the difference between M and M^*, $(A_1 + B_0 - 2.A_1.B_0)$ with a coefficient will be observed in the remainder. Therefore, whenever $a + b - 2.a.b$ pattern is seen in the remainder and there is an OR gate with inputs (a, b) in the implementation, we can conclude that the OR gate is the source of error and it should be replaced with an AND gate. Table II shows the patterns that will be observed for mis-placement 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.

TABLE II
REMAINDER PATTERNS CAUSED BY GATE MISPLACEMENT ERROR

Suspicious Gate	Appeared Remainder's Pattern	Solution
AND (a,b)	P_1 : -a-b+2.a.b	S_1 : OR (a,b)
	P_2 : -a-b+3.a.b	S_2 : XOR (a,b)
OR (a,b)	P_1 : a+b-2.a.b	S_1 : AND (a,b)
	P_2 : a.b	S_2 : XOR (a,b)
XOR (a,b)	P_1 : a+b-3.a.b	S_1 : AND (a,b)
	P_2 :-a.b	S_2 : OR (a,b)

From Section IV-B, 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 in first level from G are considered and the remainder is scanned to check whether one of the patterns in Table II 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 4: Consider the faulty circuit shown in Example 3. 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 $2, 3, 4, 6$. As we can see, remainder R does not contain any patterns shown in Table II. 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 proof. By considering $M = A_1.B_0$ and $N = A_0.B_1$, R will be rewritten as $R^* = 4.(M+N-2.M.N)$ (the GCD of coefficients of remainder terms is computed and the remainder is divided over GCD or signal's weight is computed as shown in [15]). Now, we consider the gates in the second level. This time R^* has one of the patterns shown in the Table II. Based on Table II, 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 first level gate g_i; if g_i is the buggy gate, one of the patterns in Table II 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 II (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 upper levels gets output of g_i as one of its inputs, the expression of g_i can be used instead of its output variables). As we want to compute patterns such that they contain just primary inputs (weight of gates' output is computed based on [15]), we use a dictionary to keep the expression of the gate output based on the primary inputs (line 19). The process continues until the bug is detected or set G is empty. As, suspicious gates form a cone format, when 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 Detection/Correction

```
1:  procedure BUG–CORRECTION
2:      Input: Suspected 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 ∈ G from level j do
7:              (a, b) = inputs(g_i)
8:              if !( each of (a, b) are from PI) then
9:                  a = dic.get(a)
10:                 b = dic.get(b)
11:             P_1 = ComputeP_1(a, b)
12:             P_2 = ComputeP_2(a, b)
13:             if (P_1 is found in R) then
14:               return gate g_i and solution S_1 from Table II
15:             else if (P_2 is found in R) then
16:               return gate g_i and solution S_2 from Table II
17:             else
18:                 remove g_i from G
19:                 dic.add(output(g_i), Expression(g_i(a, b)))
```

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

V. Experiments

A. Experimental Setup

The directed test generation, bug localization and bug detection algorithms were implemented in a java program and experiments were conducted on a Linux PC with Intel Processor core i5 CPU and 8 GB memory. We have tested our approach on both pre- [11] and post-synthesized gate-level arithmetic circuits that implement adders and multipliers. Post-synthesized designs were obtained by synthesizing high level description of arithmetic circuits using Xilinx synthesis tool. We consider gate misplacement 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 [11]. 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 [1] as well as random test generation. We compared our debugging results with most recent work in this context [15]. We use the benchmarks obtained from the authors [15]. To enable fair comparison, similar to [15], we randomly inserted bugs (gate changes) in the middle stages of the circuits.

B. Results

Table III presents results for test generation, bug localization and debugging methods using multipliers and adders. The first column indicates the types of benchmarks. The second and third columns show 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 indicates results for directed test generation method presented in [1] by using SMV model checker [16] (We give the model checker the advantage of knowing the bug). The fifth column represents results of random test generation method (time to generate first counterexample using random technique). The sixth column represents the time of our test generation method that generates multiple tests. As it can be observed from Table III, our method has improved directed test generation time by several orders of magnitude. The seventh column shows the CPU time for bug localization algorithm. The eight column shows the debugging time of [15] using our implementation in Java. The next column provides CPU time of our proposed approach which is the summation of test generation (TG), bug localization (BL) and debugging/correction (DC) time. The last column shows the improvement provided by our debugging framework. Clearly, our approach is an order-of-magnitude faster than the most closely related approach [15], specially in 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 in test generation of the first row (post-synthesized multiplier with 4-bit operands) of Table III, the reported results are the average of the numbers shown in Table IV.

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

Faults	[1]	Ran. tests(ms)	#tests	Faulty outputs	# Ran.	Our TG(s)
$XOR \rightarrow AND$	1.48	47.70	18	Z_7, Z_6, Z_5, Z_4	2632	0.01
$XOR \rightarrow OR$	2.12	25.95	4	Z_2	2945	0.01
$XOR \rightarrow AND$	1.95	19.21	128	Z_4	2292	0.01
$XOR \rightarrow OR$	2.27	26.43	12	Z_6, Z_5, Z_4, Z_3	2945	0.05
$XOR \rightarrow AND$	1.03	16.31	14	Z_6, Z_5, Z_4, Z_3, Z_2	2369	0.02
$AND \rightarrow XOR$	2.44	0.47	3	Z_6, Z_5, Z_4, Z_3, Z_2	1881	0.01
$AND \rightarrow OR$	2.20	1.90	2	Z_7, Z_6, Z_5	2258	0.01
$AND \rightarrow XOR$	0.89	44.17	148	Z_7, Z_6, Z_5, Z_4	2164	0.03
$OR \rightarrow AND$	2.52	11.51	148	Z_6	2920	0.01
Average	1.88	21.52	53	-	2489.55	0.01

Table IV presents the debugging results of 4-bit post-synthesized multiplier. The first column shows a possible set of gate mis-placement faults. Time to generate the first counterexample using [1] and random techniques is reported in second and third columns, respectively. The fourth column shows the number of directed tests generated by our approach

TABLE III
DEBUGGING RESULTS OF ARITHMETIC CIRCUITS . TO = TIMEOUT AFTER 3600 SEC; MO = MEMORY OUT OF 8 GB

Benchmark			Test Generation (TG)			Bug Localization (BL)	Debugging/Correction (DC)		
Type	Size	# Gates	[1] (s)	Random(s)	Our TG(s)	Bug Loc.(s)	[15]	Our (TG+BL+DC)	Improvement
post-syn. Multipliers	4	72	1.88	0.02	0.01	0.001	0.2	0.02	10x
	16	1632	42.69	1.48	0.32	0.03	7.92	1.99	3.97x
	32	6848	205.66	3.03	0.88	0.84	32.56	7.33	4.44x
	64	28K	MO	16.97	1.85	5.77	239.13	31.37	7.62x
	128	132K	MO	66.52	3.11	34.38	2148.96	271.89	7.90x
	256	640K	MO	TO	16.44	157.19	TO	1002.73	-
pre-syn. Multipliers	4	94	1.27	0.04	0.01	0.001	0.2	0.04	5x
	16	1860	43.11	1.93	0.41	0.03	8.12	1.89	4.26x
	32	7812	189.50	5.69	0.97	0.65	31.64	9.8	3.22x
	64	32K	MO	29.07	2.01	3.10	207.50	26.88	7.71x
	128	129K	MO	83.60	3.18	23.33	2001.41	178.28	11.22x
	256	521K	MO	TO	19.31	118.51	TO	994.64	-
post-syn. Adder	64	573	154.97	1.51	0.67	0.01	2.12	0.82	2.49x
	128	1251	MO	3.48	1.25	0.05	5.33	1.82	2.92x
	256	2301	MO	10.64	3.78	0.14	12.66	5.74	2.20x
pre-syn. Adder	64	444	128.12	1.15	0.31	0.01	1.55	0.57	2.71x
	128	880	MO	4.40	0.84	0.03	3.73	1.45	2.57x
	256	1698	MO	9.10	2.09	0.28	8.09	3.94	2.05x

to activate the bug (each of them activates the bug). The fifth column lists the outputs that are affected by the fault (activated by the respective tests reported in the fourth column). The sixth column shows the number of random tests required to cover all of our directed tests. It demonstrates that even for such small circuits, using random tests to activate the error is impractical. The last column shows our test generation time. As mentioned earlier, the average of these scenarios is reported in the first row of Table IV.

The experimental results demonstrated 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 misplacement 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.

VI. CONCLUSION

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. We used the remainder produced by equivalence checking methods to generate directed tests that are guaranteed to activate the source of the bug when 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 detect and correct the bug. 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.

VII. ACKNOWLEDGMENTS

This work was partially supported by the NSF grants (CCF-1218629 and CNS-1441667) and SRC grant (2014-TS-2554). We would like to thank Prof. Maciej Ciesielski from the University of Massachusetts at Amherst for providing their functional extraction framework [11].

REFERENCES

[1] M. Chen and P. Mishra, "Functional Test Generation using Efficient Property Clustering and Learning Techniques," in *IEEE Trans. on CAD* 2010.
[2] M. Chen, X. Qin, H. Koo, P. Mishra, *System-level Validation - high-level modeling and directed test generation techniques*. Springer, 2012.
[3] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Vinov and a. Vinov, "Genesys-pro: Innovations in test program generation for functional processor verification," *Design and Test of Computers*, 2004.
[4] L. Liu and S. Vasudevan, "Efficient validation input generation in RTL by hybridized source code analysis," in *DATE* 2011.
[5] R.E. Bryant and Y.A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *DAC* 1995.
[6] H. Mangassarian, A. Veneris, S. Safapour, M. Benedetti and D. Smith, "A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test," in *ICCAD* 2007.
[7] B. Le, H. Mangassarian, B. Keng and A. Veneris, "Non-solution implications using reverse domination in a modern sat-based debugging enviroment," in *DATE*, 2012.
[8] K. Chang et al., "Accurate rank ordering of error candidates for efficient HDL design debugging," in *TCAD* 2009.
[9] K. Chang, I. Markov and V. Bertacco, "Equivalence verification of polynomial datapaths using ideal membership testing," in *TCAD* 2006.
[10] F. Farahmandi and B. Alizadeh, "Grobner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," in *Microprocessor and Microsystems* 2015.
[11] M. Ciesielski, C. Yu, W. Brown, D. Liu and A. Rossi, "Verification of gate-level arithmetic circuits by function extraction," in *DAC* 2015.
[12] X. Sun, P. Kalla, T. Pruss and F. Enescu, "Formal verification of sequential Galois field arithmetic circuits using algebraic geometry," in *DATE*, 2015.
[13] D. Cox, J. Little and D. O'Shea, *Ideals, varieties, and algorithms*. Springer, 1997.
[14] O. Wienand, M. Welder, D. Stoffel, W. Kunz and G.M. Greuel, "An algebraic approach for proving data correctness in arithmetic data paths," in *CAV* 2008.
[15] S. Ghandali and C. Yu and W. Brown and M. Ciesielski, "Logic debugging of arithmetic circuits," in *ISVLSI* 2015.
[16] *SMV Model Checker*, http://www.kenmcmil.com.
[17] J. Lv, P. Kalla and F. Enescu, "Efficient Groebner basis reductions for formal verification of galois field multipliers," in DATE 2012.
[18] X. Guo, R.G. Dutta, Y. Jin, F. Farahmandi and P. Mishra, "Pre-Silicon Security Verification and Validation: A Formal Perspective," in *DAC* 2015.