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 remainder 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.

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 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 are 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 Field 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 $f_{\text{spec}}$. 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 $\{a_i, b_i, z_i\} \in \mathbb{Z}_2$. 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: $(2^{n+m-1}.z_{n+m-1} + \ldots + 2.z_1 + z_0) - (2^{n-1}.a_{n-1} + \ldots + 2.a_1 + a_0) = 0$.

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 $\text{NOT, AND, OR, XOR}$ gates.

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

The verification method is based on transforming $f_{\text{spec}}$ 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 $f_{\text{spec}}$. The verification process starts from $f_{\text{spec}}$ 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 $f_{\text{spec}}$ 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.

\[
\begin{align*}
\text{step}_3(\text{remainder}): & 2.A_1 + 2.B_0 - 4.A_1.B_0.
\end{align*}
\]  

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-

![Fig. 2. Faulty gate-level netlist of a 2-bit multiplier](image-url)
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. 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 get remainder R polynomial and primary inputs (PIs) in the remainder as inputs and feeds binary values to PIs and computes the total value of a term. If the summation (value) of all the terms is non-zero, the corresponding primary input assignments are added to the set of Tests.

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 \in R do |
| 6: if (T_j(s_i)) then |
| 7: Sum += C_{T_j} |
| 8: if (Sum \neq 0) then |
| 9: Tests = Tests \cup s_i |
| return Tests |

Example 2: Consider the faulty circuit shown in Fig. 2 and the remainder polynomial \( R = 2(A_1 + B_1 - 2A_1B_0) \). 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 tests are shown in Table I.

Table I: Directed Tests to Activate Fault Shown in Fig. 2

| 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 \( E = \{e_1, e_2, \ldots, e_n\} \). 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 \( e_i \in E \) do |
5: find cones that construct \( e_i \) and put in \( C_{e_i} \) |
6: \( C_S = C_{e_0} \) |
7: for \( e_i \in E \) do |
8: \( C_S = C_S \cap C_{e_i} \) |
9: return \( C_S \) |

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 \( C_{e_i} = \{e_1, e_2, \ldots, e_n\} \) that constructs the value of each \( e_i \) from set \( E \) (line 4-5). These cones contain suspicious gates. We intersect all of the suspicious cones \( C_{e_i} \) to prune the search space and improve the efficiency of bug localization algorithm. The intersection of these cones are stored in \( C_S \) (line 7-8).

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 \( R = 4A_1B_0 + 4A_0B_1 - 8A_0A_1B_0B_1 \). 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: \( C_{Z_2} = \{2, 3, 6, 7\} \) and \( C_{Z_3} = \{2, 3, 6, 4, 8\} \). The intersection of the cones that produce faulty outputs is \( C_S = \{2, 3, 6\} \). As a result, gates \( \{2, 3, 6\} \) are potentially responsible as a source of the error.

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 \( R = 4.A_1.B_0 + 4.A_9.B_1 - 8.A_9.A_1.B_0.B_1 \) and the potentially faulty gates are numbered 2, 3, 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 and 3 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 and 3 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 \( N = A_9.B_1 \), \( R \) 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; based on their levels (lowest level first)
5: for each level j do
6: for each g; ∈ G from level j do
7: (a, b) = inputs(g;)
8: if !(each of (a, b) are from Pl) then
9: a = dic.get(a)
10: b = dic.get(b)
11: P1 = ComputeP1(a, b)
12: P2 = ComputeP2(a, b)
13: if (P1 is found in R) then
14: return gate g; and solution S1 from Table II
15: else if (P2 is found in R) then
16: return gate g; and solution S2 from Table II
17: else remove g; from G
18: dic.add(output(g;), Expression(g;(a, b)))

Example 5: We want to apply Algorithm 3 on the case shown in Example 4. We start from gate 2 and compute P1 = −A1 − B0 + 2. A1. B0 and P2 = −A1 − B0 + 3. A1. B0 for gate 2. As these patterns do not exist in the remainder, gate 2 is correct and the dictionary will be updated as (M = A1. B0). The same will happen for gate 3 and dictionary will be updated as (M = A1. B0, N = A0. B1) at the end of this iteration. Now, the gate 6 is considered and the P; s are as follows: P1 = A1. B0 + A0. B1 − 2. A1. B0. A0. B1 and P2 = A1. B0. A0. B1. Considering that R = 4(A1. B0 + 4. A0. B1 − 2. A0. A1. B0. B1), P2 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 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.

Table IV presents the debugging results of 4-bit post-synthesized multiplier with 8-bit operands and 4-bit operators. Table IV shows the number of directed tests generated by our approach can be generated independently 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 used 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 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 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

<table>
<thead>
<tr>
<th>Bench</th>
<th># Gates</th>
<th>Size</th>
<th>Faulty Gates</th>
<th># Runs</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR → AND</td>
<td>244</td>
<td>36.30</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>255</td>
<td>36.40</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>244</td>
<td>36.30</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>255</td>
<td>36.40</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>244</td>
<td>36.30</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>255</td>
<td>36.40</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>244</td>
<td>36.30</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>255</td>
<td>36.40</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>244</td>
<td>36.30</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>255</td>
<td>36.40</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>244</td>
<td>36.30</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>255</td>
<td>36.40</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>244</td>
<td>36.30</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>255</td>
<td>36.40</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>244</td>
<td>36.30</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>255</td>
<td>36.40</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>244</td>
<td>36.30</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>255</td>
<td>36.40</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>244</td>
<td>36.30</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
<tr>
<td>XOR → AND</td>
<td>255</td>
<td>36.40</td>
<td>14</td>
<td>63</td>
<td>63</td>
<td>1</td>
<td>101</td>
<td>0.10</td>
</tr>
</tbody>
</table>

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.
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