Automated Debugging of Arithmetic Circuits using Incremental Gröbner Basis Reduction

Farimah Farahmandi and Prabhat Mishra
Department of Computer and Information Science and Engineering
University of Florida, USA

***Abstract*—Symbolic algebra is a promising approach to verify large and complex arithmetic circuits. Existing algebraic-based verification methods generate a remainder to indicate buggy implementation. The remainder is beneficial for debugging of the faulty implementation since it can be used for automated test generation, bug localization, and bug correction. However, existing equivalence checking approaches are not scalable and lead to explosion in size of the remainder when the design is faulty. To make the matters worse, the location of the bug can also lead to the explosion in the number of remainder terms. In this paper, we propose an incremental equivalence checking method to address the scalability challenges by solving the verification problem in the increasing order of design's input complexity. Our proposed approach makes two important contributions. It is able to generate smaller and compact remainders for large designs. Our proposed incremental debugging is capable of localizing and correcting hard-to-detect bugs irrespective of their location in the design. Experimental results demonstrate that our approach can efficiently debug most difficult bugs in large arithmetic circuits when the state-of-the-art methods fail.**

I. Introduction

Arithmetic circuits are critical components in most of the modern designs due to their application in computation-intensive tasks such as multimedia and signal processing. They are also used in security-related hardware to implement cryptography operations. Increasing demand for fast and accurate arithmetic components has led to several non-standard and optimized implementations that are susceptible to faults. The verification and debugging of arithmetic circuits are complex due to bit-blasting and custom transformation of the implementation based on different arithmetic and logical relations. Similar challenges are also highlighted in recent industrial studies [1]. Therefore, designers are seeking for efficient and automated verification and debugging approaches.

Traditional automatic verification and debugging approaches are mostly based on simulation, decision diagrams such as BDDs, BMDs [2], reverse engineering [3] and SAT solvers [4]. They all suffer from scalability problems. Recent verification and debugging approaches utilize symbolic computer algebra [5], [6], [7], [8], [10] as shown in Figure 1. These methods convert the gate-level implementation to a set of polynomials and check whether implementation polynomials can construct the functionality of the specification polynomial. These methods generate a remainder if the implementation and specification are not equivalent. These approaches are promising to verify large arithmetic circuits when the implementation is correct, or bugs exist very close to the primary inputs. However, existing methods fail to verify buggy arithmetic circuits when the bug is in the deeper stages of the design

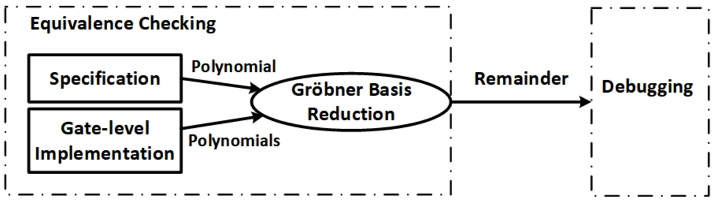

Fig. 1. Overview of existing equivalence checking and debugging approaches.

or when there are multiple bugs. Since the remainder (either zero or non-zero) defines the result of the equivalence checking process, challenges in remainder generation raise questions about scalability and applicability of existing approaches. On the other hand, the produced remainder is beneficial to efficiently debug faulty arithmetic circuits since it can be used to generate directed tests to activate unknown bugs (if any) in the implementation. Remainder generation is the first step of the verification process. Therefore, unavailability of the remainder implies that subsequent steps (test generation, bug localization, and bug correction) cannot be performed.

Depending on the location of the bug, the remainder generation can be challenging. The existence of a bug in the deeper stages of the design may make it extremely difficult to generate the remainder due to an explosion in the number of remainder terms (we refer this as *term explosion effect*). The reason is that the faulty gate may introduce new terms during the intermediate steps of the specification polynomial's reduction. These extra terms are multiplied to polynomials of other gates and grow continuously until the remainder contains only primary inputs, leading to an explosion in the number of remainder terms.

Figure 2 compares the number of terms in different iterations of correct and buggy implementations. As it can be observed from Figure 2, the number of terms drastically grows when the bug is in the deeper stages of the design. If the number of remainder terms for a simple 4×4 multiplier grows that quickly, it is impossible to deal with the term explosion effect in case of buggy and complex arithmetic circuits. Moreover, having several bugs in the implementation as well as dealing with complex designs can make the remainder generation infeasible (in the worst case, a remainder can contain $n!$ terms where n is the number of primary inputs). The debugging approach presented in [9] performs well when the bug is close to the primary inputs. However, it is expected to fail if the bug is in the deeper stages of the design due to term explosion effect. In other words, if existing approaches cannot generate a remainder in a reasonable time, they are not useful for debugging arithmetic circuits. We propose an incremental equivalence checking method to enable fast and compact remainder generation.

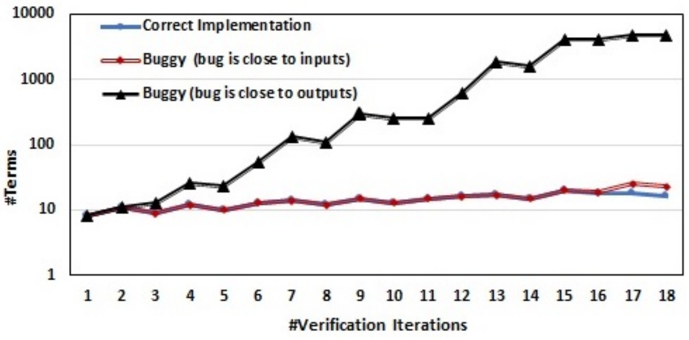

Fig. 2. Comparison of the number of terms in different iterations in verification of a 4x4 multiplier when: i) the implementation is correct, ii) the implementation is buggy and the bug is close to the primary inputs, and iii) the implementation is buggy implementation and the bug is in the deeper stages of the design (e.g., close to the primary outputs).

In this paper, we address the above challenges by proposing an incremental debugging framework. The proposed approach partitions the primary inputs' space of the design in order to solve the verification and debug problems in the increasing order of the design complexity. The verification test is decomposed to n independent equivalence checking problems and it is performed in several iterations where the equivalence of the specification and implementation is checked with considering primary inputs' constraints. The basic intuition behind our work is to observe the fact that it is efficient to debug an error in a smaller region (e.g., the portion of the design that multiples the first two bits) instead of searching in the whole 4x4 multiplier. If no bugs found in the region representing 1×1 multiplication, in the next iteration a larger region (e.g., representing 2×2 multiplication) will be searched. On the surface, it may seem that our approach will take longer than solving directly on the original design, but as proposed work (Section IV) and results (Section V) demonstrate that our well-crafted incremental approach drastically reduces the debugging complexity.

In each iteration, specification and implementation polynomials are updated based on the primary inputs' constraints, and Gröbner basis reduction is used to generate a remainder in order to define the result of the verification. If the verification results in a non-zero remainder, the implementation is buggy. We use the generated remainder as well as inputs' constraints to detect and correct the source of the error. Using the incremental verification approach enables us to efficiently generate a remainder for a faulty design. Our experimental results demonstrate that our approach improves the performance of existing debugging approaches by several orders of magnitude.

This paper makes three important contributions: i) develops a scalable framework for incremental equivalence checking using Gröbner basis reduction, ii) enables incremental debugging of hard-to-detect bugs, and iii) enables detection and fixing of complex bugs when existing state-of-the-art techniques fail.

The rest of the paper is organized as follows. We discuss related work in Section II. Section III provides an overview about existing equivalence checking based on symbolic algebra. Section IV describes our proposed incremental equivalence checking and debugging framework. Section V presents our experimental results. Section VI concludes the paper.

II. Related Work

Traditional approaches of automated debugging of arithmetic circuits rely on simulation-based techniques as well as decision diagrams [2]. To address the scalability challenges of these techniques, debugging approaches based on SAT solvers were proposed. These approaches are based on either inserting logic corrector components in the implementation [4], using abstraction and refinements [11], [12] or using Quantified Boolean Formula [13]. They model the circuit using CNF model, and a SAT solver is used to localize the sources of error. The success of these approaches is dependent on the performance of SAT solvers, and they fail for large and complex arithmetic circuits. There are some approaches based on SMT solvers to find a counterexample and localize the bug of the faulty implementation [14], [15]. However, these approaches suffer from the required manual interventions and cannot handle large designs. Model checking based methods suffer from challenges in assertion generation as well as state space explosion [16], [17], [18]. The idea of incremental checking of the specification to the implementation at intermediate comparison points instead of considering a monolithic verification problem is not new. Techniques based on induction and term rewriting is presented for verification of arithmetic circuits [19], [20]. These methods use a theorem prover as well as a database of rewrite rules to incrementally verify the design. However, these methods suffer from manual interventions and do not address the debugging problem.

Ghandali et. al [21] proposed an automated debugging approach based on symbolic computer algebra which scans the entire implementation to find and fix the bug in the design. This approach suffers from scalability concerns. Farhamandi and Mishra [9] generated directed tests to activate any unknown bug and localize the source of the error. While these approaches [21], [9] are capable of fixing the unknown bug, their effectiveness is dependent on the result (generated remainder) of the equivalence checking techniques for buggy designs. None of the existing techniques [6], [8], [22] are capable of generating a remainder when the bug is deep inside the design. In this paper, we propose an incremental equivalence checking technique which enables efficient debugging by generating a compact remainder for hard-to-detect bugs.

III. Background and Motivation

Equivalence checking of arithmetic circuits against their word-level specifications can be performed using symbolic algebra [8], [5] that maps equivalence checking problem of an arithmetic circuit to ideal membership testing. In algebraic domain, ideal membership testing refers to checking whether a polynomial resides inside a given ideal. In hardware equivalence checking using symbolic algebra, the specification of the circuit is modeled as a polynomial f_{spec}, and the implementation is converted to a set of polynomials, which is used to construct ideal I. In order to check the equality between the specification and the implementation, f_{spec} is checked to figure out whether it belongs to ideal I. The equivalence checking based on symbolic algebra inherits advantages from the word-level abstraction of the specification of an arithmetic circuit as well as its implementation.

Consider field $\mathbb{K}$ where $\mathbb{K}[x_1, x_2, ..., x_n]$ is a polynomial ring with variables $x_1, x_2, ..., x_n$. Let $M = {x_1}^{\alpha_1} * {x_2}^{\alpha_2} * ... * {x_n}^{\alpha_n}$ be a monomial where $\{\alpha_1, \alpha_2, ..., \alpha_n\}$ are non-negative integers. $f = c_1 * M_1 + c_2 * M_2 + ...c_d * M_d$ is called polynomial in ring $\mathbb{K}[x_1, x_2, ..., x_n]$ where $c_1, c_2, ..., c_d$ are coefficients and $M_1, M_2, ..., M_d$ are monomials. A monomial order ">" is defined over a set of monomials in a polynomial such that the monomial set has the smallest element under order $>$ and if monomial $M_i > M_j$, we can conclude $M_i + M_s > M_j + M_s$ with respect to $>$. Considering order $>$ and polynomial f, $LM(f)$ shows the largest monomial of f, $LC(f)$ shows the coefficient of $LM(f)$ and $LT(f)$ shows the leading term of polynomial f which is equal to: $LT(f) = LM(f) * LC(f)$. Set $F = \{f_1, f_2, ..., f_s\}$ constructs an ideal $I =< f_1, f_2, .., f_s >= \{\sum_{i=1}^{s} h_i * f_i : h_i, f_i \in \mathbb{K}[x_1, x_2, ..., x_n]\}$. In other words, set F is called the generator (basis) of ideal I. An ideal can have several basis. One of the important basis is called Gröbner basis (G) which can solve the ideal membership problem. To test whether f belongs to ideal I, polynomial division is deployed. Polynomial f is reducible by polynomial g_j if leading term of f is divisible by leading term g_j ($r = f - \frac{lt(f)}{lt(g_j)} * g_j$). Similarly, polynomial f can be reduced over set G as $f_i \xrightarrow{G}_+ r$. If set G is Gröbner basis of ideal I and reduction of f over set G generates zero remainder, polynomial f resides in ideal I. Otherwise, f does not belong to ideal I. Gröbner basis can be computed by applying Buchberger algorithm [23]. Generator F can be also Gröbner basis of Ideal I with respcet to $>$ when all of the polynomials in set F have relatively prime leading terms [10]. In this case, Gröbner basis does not need to be computed and $F = G$.

The arithmetic circuit equivalence checking formulation starts with converting the design specification to a polynomial f_{spec} using primary inputs and primary outputs as variables. f_{spec} shows the word-level abstraction of the functionality of an arithmetic circuit. For instance, the specification of a n-bit multiplier with primary inputs $A = \{a_0, a_1, ..., a_{n-1}\}$ and $B = \{b_0, b_1, ..., b_{n-1}\}$ and primary output $Z = \{z_0, z_1, ...z_{2n-1}\}$ can be formulated as $Z = A * B$ or can be written as $(2^{2n-1} * z_{2n-1} + ... + 2 * z_1 + z_0) - (2^{n-1} * a_{n-1} + ... + 2 * a_1 + a_0) * (2^{n-1} * b_{n-1} + ... + 2 * b_1 + b_0) = 0$ where $a_i, b_i, z_i \in \{0, 1\}$.

Similarly, the gate-level implementation can be modeled as a set of polynomials (F) by converting each gate to one polynomial as shown in Equation 1. Each variable x_i shows the gate input and each variable y_i shows the gate output where both input and output can get either zero or one values (decimal) and ${x_i}^2 = x_i$ (same for y_i).

$$\begin{aligned}
&y_1 = \neg x_1 \rightarrow f_{not} : y_1 - (1 - x_1) = 0, \\
&y_2 = x_1 \wedge x_2 \rightarrow f_{and} : y_2 - x_1 * x_2 = 0, \\
&y_3 = x_1 \vee x_2 \rightarrow f_{or} : y_3 - x_1 + x_2 - x_1 * x_2 = 0, \\
&y_4 = x_1 \oplus x_2 \rightarrow f_{xor} : y_4 - x_1 + x_2 - 2 * x_1 * x_2 = 0
\end{aligned} \tag{1}$$

Implementation polynomials ($F = \{f_1, f_2, ..., f_s\}$) are in the form of $\mathbb{Z}_2[x_1, x_2, ..., x_n]/\langle x_1 - {x_1}^2, x_2 - {x_2}^2, ..., x_n - {x_n}^2\rangle$. Set F constructs an ideal $I =< f_1, f_2, .., f_s >$ over Boolean ring $\mathbb{Z}_2$. Implementation polynomials are derived based on Equation 1 and the topological order of the circuit where the primary inputs have the lowest values and the primary outputs have the highest values is considered. For example, leading term of f_{not} is equal to y_1 ($LM(f_{not}) = y_1$). Since combinational arithmetic circuits are acyclic and each output variable appears in the polynomials once, polynomials in set F have relatively prime leading terms. Therefore, set F is itself Gröbner basis of ideal I. The equivalence checking of the specification and the implementation maps to test whether f_{spec} belongs in ideal I. Polynomial division is deployed to test the membership of f_{spec}. The equivalence checking starts with consecutively reducing the f_{spec} over implementation polynomials (F) until it leads to either zero remainder or a remainder that contains only primary inputs. If the remainder is zero, it shows that the arithmetic circuit correctly implemented the specification. However, a non-zero remainder indicates that the implementation is faulty.

Example 1: Suppose that our goal is to verify a 2x2 multiplier shown in Fig. 3. Suppose that, the OR gate with inputs (A_0, B_0) has been incorrectly used instead of an AND gate. The specification of a 2-bit multiplier is shown by f_{spec}. 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 for Gröbner basis reduction. The verification procedure starts with reducing f_{spec} and its terms over implementation polynomials as shown in Equation 2. For instance, term $4 * Z_2$ from f_{spec} is replaced by polynomial $4*(R+O-2*R*O)$ since gate 7 constructs the implementation polynomial $Z_2-(R+O-2*R*O) = 0$. Note that, terms in the same level are reduced together. The non-zero remainder indicates that the implementation is not correct. The remainder will become zero if gate 1 is replaced by an AND gate and equivalence checking procedure is repeated. ■

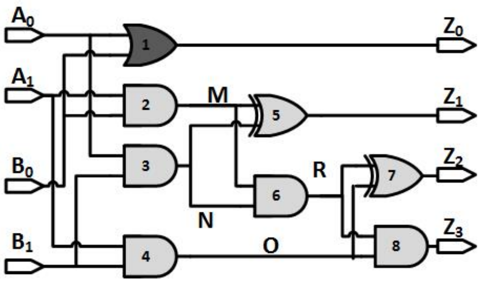

Fig. 3. Gate-level netlist of a faulty 2x2 multiplier.

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

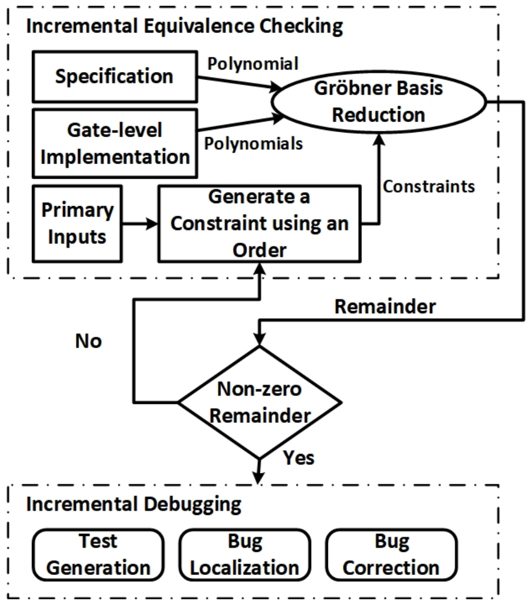

Fig. 4. Overview of our proposed approach. Our overall flow consists of three different parts: i) partitioning the inputs' space into different constraints based on a selected inputs' order, ii) remainder generation using incremental equivalence checking; and iii) incremental debugging of the design.

IV. Efficient Debugging of Arithmetic Circuits

In this paper, we present an approach to incrementally perform equivalence checking between an arithmetic circuit specification and its implementation. *We consider gate misplacement that changes the functionality of the design as our fault model.* Figure 4 shows an overview of our proposed approach. Figure 4 highlights three key parts of our approach: i) partition the inputs' space into different constraints based on a selected inputs' order; ii) incremental equivalence checking; and iii) incremental debugging approach. The rest of this section describes these steps in detail.

A. Remainder Generation using Incremental Equivalence Checking

In this section, we present an approach to solve the equivalence checking problem in complex arithmetic circuits incrementally. The proposed approach is based on partitioning the input space of the design by applying certain constraints on primary inputs to solve the equivalence checking problem for each input constraint. If set $\mathbb{M} = \{0,1\}^n$ shows all input combinations of a design with input bits $\{x_0, x_1, ..., x_{n-1}\}$ and if specification ($\mathbb{S}$) and implementation ($\mathbb{I}$) are equivalent for all combinations of ($\mathbb{S} \overset{\mathbb{M}}{\equiv} \mathbb{I}$), they should also be equivalent for any input combinations that belongs to $\mathbb{M}$ ($\forall M \subset \mathbb{M}, \quad \mathbb{S} \overset{M}{\equiv} \mathbb{I}$). If the implementation is buggy, at least one of the intermediate reductions will result in a non-zero remainder.

Clearly, it is not feasible to repeat the equivalence checking procedure for all 2^n input combinations if the design contains n bits of primary inputs. In the existing methods, the inputs are represented by abstract symbols that can get any values. However, we propose an input space partitioning method that is based on keeping some of the primary inputs in symbolic form and assigning Boolean values (either zero or one) to the rest of the primary inputs. This approach expedites the remainder generation time, and it also reduces the number of terms in the remainder and makes it possible to generate a remainder irrespective of the location of the bug. In other words, this approach prevents the remainder's term explosion effect.

Algorithm 1 shows our input partitioning approach. Given the set of primary inputs K with a particular order the algorithm returns n different constraints on primary inputs where n is the number of primary inputs. Initially, the algorithm sets all of the inputs to zero except the first input in set K which is kept in the symbolic form, and the algorithm adds them to the set of results $\mathbb{M}$ (lines 5-8). In the next step, it keeps the first input in the symbolic form and sets the second input of the ordered set as '1', and sets other inputs to '0', and adds the constraints to the result. This process continues until all of the inputs are kept in their symbolic form except the last one which is set to true. The variable $index$ presents the index of primary inputs that should be assigned to true (line 11). The variables before the index variable are kept in their symbolic form, and variables that come after the index are assigned to false (lines 12-15). In each iteration, the $index$ variable is updated (line 16). The algorithm returns the set of constraints as output. This algorithm guarantees (see the proof of Theorem 1) that the entire inputs' space is covered since all of the combinations of primary inputs are considered (each input bit is assigned to either one, zero or kept in the symbolic form which can take both values).

Algorithm 1 Generation of Input Constraints

```
1:  procedure INPUT–CONSTRAINTS–GENERATOR
2:      Input: Primary inputs $K$
3:      Output: Set of Constraints Map $\mathbb{M}$
4:      new map $M = \{\}$; $n = sizeOf(K)$
5:      $M.add(0, K[0])$
6:      for $i = 1; i < n; j++$ do
7:          $M.add(i, false)$
8:      $\mathbb{M}.add(M)$, $index = 1$
9:      for $i = 0; i \leq n; i++$ do
10:         $M = \{\}$
11:         $M.put(index, true)$
12:         for $j = 0; j < index; j++$ do
13:             $M.add(j, K[j])$
14:         for $j = index + 1; i \leq n; j++$ do
15:             $M.put(j, false)$
16:         $index++$
17:         $\mathbb{M}.add(M)$
        return $\mathbb{M}$
```

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

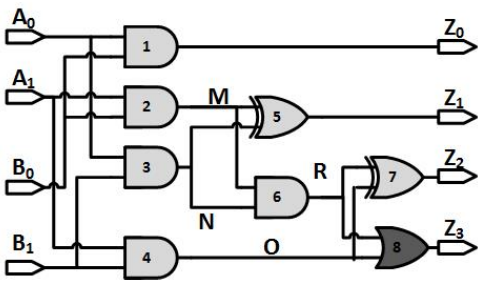

Fig. 5. Faulty netlist with one bug (gate 8 should have been an AND)

TABLE I
INPUT CONSTRAINTS TO EFFICIENTLY VERIFY AND DEBUG FAULTY CIRCUIT SHOWN IN FIGURE 5.

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

Theorem 1. *A constraint table with n variables (n rows) effectively captures 2^n input sequences.*

Proof. The first row of the constraint table covers two of the input sequences since the first variable is kept as its symbolic form (which it can be either 0 or 1) and other variables are assigned to 0s. Similarly, the second row also covers two combinations as the first variable is in its symbolic form and the second variable is fixed to 1 and the rest of the variables are assigned to zero. Likewise, in the row i where $i \neq 1$, i variables are in their symbolic forms and one variable is assigned to 1 and rest of the variables are assigned to 0. Therefore, row $i \neq 1$ captures 2^{i-1} input sequences. Therefore, for n ($n > 1$) rows we have:

$$2 + \sum_{i=2}^{n} 2^{i-1} = 2 + (2^n - 2) = 2^n$$

We propose an incremental equivalence checking method using the constraints computed based on Algorithm 1. The original equivalence checking problem is mapped to n equivalence checking sub-problems where the specification and implementation polynomials are updated by applying the corresponding constraints. In each sub-problem, a new set of implementation polynomials is computed based on propagating the integer values of the corresponding constraint and considering them while constructing polynomials of each gate and each fanout-free region. Specification polynomial is also updated by applying the conditions of primary inputs in the original specification polynomial. In each sub-problem, the corresponding specification polynomial is reduced over the related implementation polynomials. If the remainder is non-zero, the given constraint manifests some bugs in the design. The implementation and specification of an arithmetic circuit are equivalent if remainders of each of the n sub-problems is computed as a zero remainder.

Algorithm 2 shows the procedure of incremental equivalence checking for one iteration. It takes input constraints M_i, original specification polynomial f_{spec} as well as partitioned gate-level netlist C as inputs. It evaluates whether specification

Algorithm 2 Incremental Equivalence Checking Algorithm

```
1: procedure INCREMENTAL EQUIVALENCE–CHECKER
2:     Input: Input constraint $M_i$, specification polynomial $f_{spec}$, Gate-level netlist $C$
3:     Output: Remainder $r$ if the implementation is faulty
4:     $f_{spec_i}$ =findSpecificationPolynomial($f_{spec}$, $M_i$)
5:     $\mathbb{F}_i$ =findImplementationPolynomials($C$, $M_i$)
6:     $r_i$ = reduction of $f_{spec_i}$ over $f_j s \in \mathbb{F}_i$
7:     if ($r_i != 0$) then
8:         Implementation is buggy
9:         return $r_i$
       return 0  ▷ correct implementation for constraint $M_i$
```

f_{spec} and implementation C are equivalent considering the constraint M_i. The algorithm returns a counterexample in case of mismatch. Specification polynomial f_{spec} is updated based on the input constraints (line 4). Implementation polynomials corresponding to fanout-free region's of C are also reconstructed by applying the constraints. Note that, the polynomial of a fanout-free region is reconstructed if at least condition of one of the region's inputs is different. Otherwise, the polynomial computed in previous iteration is reused. The equivalence checking uses Gröbner basis reduction to reduce f_{spec_i} over implementation polynomials $\mathbb{F}_i$ to find a non-zero remainder if a bug exists in the implementation. The overall equivalence checking algorithm consists of n iterations each responsible for one inputs' constraints.

Example 3: Consider the 2-bit multiplier shown in Figure 5. We want to apply the incremental equivalence checking approach of Algorithm 2 using all of the input constraints shown in Table I to verify the correctness of the implementation. Equation 3 shows the steps of the verification. Specification and implementation polynomials are updated using each constraint. For instance, polynomial of gate 3 is computed as: $N = A_0 * B_1 = 0$ as A_0 and B_1 are considered zero in the first iteration (first row of the Table I). Since the last iteration generates a non-zero remainder, the implementation is faulty. ■

$$\begin{aligned}
&\mathbb{F}_1 = \{Z_0 = 0, M = 0, N = 0, O = 0, R = 0, Z_1 = 0, Z_2 = 0, Z_3 = 0\}\\
&f_{spec_1} : 8 * Z_3 + 4 * Z_2 + 2 * Z_1 + Z_0\\
&step_{1_1}(remainder) : 0\\
&\mathbb{F}_2 = \{Z_0 = 0, M = 0, N = 0, O = A_1, R = 0, Z_1 = 0, Z_2 = A_1, Z_3 = A_1\}\\
&f_{spec_2} : 8 * Z_3 + 4 * Z_2 + 2 * Z_1 + Z_0 - 4 * A_1\\
&step_{1_2} : 2 * Z_1 + Z_0 + 8 * A_1\\
&step_{2_2}(remainder) : 8 * A_1
\end{aligned} \tag{3}$$

Different ordering improves the complexity of the incremental verification approach and enables generation of a more efficient remainder for the same bug. Moreover, the complexity of equivalence checking using the given constraint is lower than existing approaches. The generated remainder also has lower complexity compared to with the original remainder that can be achieved with existing methods ($r = 8.A_1 * B_1 - 8 * A_1 * A_0 * B_0 * B_1$).

The merit of this approach can be observed for verifying complex and buggy implementation since the size of the remainder's terms are reduced by assigning the design variables to either zero or one. Therefore, the possibility of term explosion is drastically reduced. On the other hand, if the implementation is correct, all of the n iterations should be performed. However, the time complexity does not grow n times since the time and memory complexities of most of the iterations are negligible.

B. Incremental Debugging

The generated remainder and the corresponding constraints can be used to debug the faulty design more effectively based on the approach presented in [9]. Our approach is orthogonal to [9] and can be used on top of it. To generate directed tests to activate the existing fault, assignment to the remainder' variables can be performed such that the integer value of the remainder becomes non-zero. Smaller remainder needs less effort to generate directed tests. The generated test and associated constraints are used to find faulty outputs and localize the source of the bug. The remainder contains terms which show the difference in the functionality of the faulty gate with the expected correct gate based on the functionality of the gate's inputs. Therefore, for each suspicious gate, two patterns are constructed based on Table II to detect and correct the source of the bug. Using the incremental debugging, it is possible that some inputs of gates are equal to zero. Therefore, some suspicious AND gates may have two equal patterns during pattern construction. It can be observed from Table II, while generating patterns for a suspicious AND gate, if either input a or b is equal to zero, then P_1 and P_2 will be equal. Only in this case, the equivalence checking should be repeated in order to find the solution of the faulty AND gate (whether the correct gate should be an OR gate or a XOR gate).

TABLE II
TEMPLATES CAN BE CAUSED BY GATE MISPLACEMENT ERROR

Faulty Gate	Appeared Remainder's Pattern	Correct Gate
AND (a,b)	$P_1 : -a - b + 2 * a * b$	OR (a,b)
	$P_2 : -a - b + 3 * a * b$	XOR (a,b)
OR (a,b)	$P_1 : a + b - 2 * a * b$	AND (a,b)
	$P_2 : a * b$	XOR (a,b)
XOR (a,b)	$P_1 : a + b - 3 * a * b$	AND (a,b)
	$P_2 : -a * b$	OR (a,b)

Algorithm 3 shows an overview of our proposed incremental approach. The algorithm includes four key parts: i) choosing an efficient order of the primary inputs (line 3); ii) partitioning the inputs' space into different constraints based on the given order (line 4 is Algorithm 1); iii) incremental equivalence checking (line 6 is Algorithm 2); and iv) incremental debugging (lines 9-12) which can be performed using [9].

Example 4: Considering the faulty implementation shown in Figure 5 and remainder $r = 4 * A_1$, the only assignment that makes r non-zero is $A_1 = 1$. Considering the other constraints that generates the remainder, $A_1, B_1, A_0, B_0 =$ "$11XX$" is a directed test to activate the fault. The test activates the effect of the bug in primary output Z_3. Therefore, gates $2, 3, 4, 6, 8$ are suspicious. Patterns are constructed for each of the gate as $2(P_1 = P_2 = 2 * A_1), 3(P_1 = P_2 = 1), 4(P_1 = P_2 = 4 * A_1), 6(P_1 = P_2 = 0), 8(P_1 = 8 * A_1, P_2 = 0)$. Therefore, gate 8 is faulty and it should be replaced with an AND

Algorithm 3 Incremental debugging Algorithm

```
1:  procedure INCREMENTAL –DEBUGGING–ALGORITHM
2:      Input: Specification polynomial $f_{spec}$, Gate-level netlist $C$, Primary inputs $PI$
3:      Output: Potential faulty gate and its solution
4:      $K$ = OrderPrimaryInputs(PI)
5:      $\mathbb{M}$=GenerateInputConstraints(K)
6:      for each input constraints $M_i \in \mathbb{M}$ do
7:          $r_i$=IncrementalEquivalenceChecking($M_i, f_{spec}, C$)
8:          if ($r_i != 0$) then
9:              $\mathbb{T}$=GenerateDirectedTests($r_i$)
10:             $\mathbb{G}$=BugLocalization($\mathbb{T}, C$)
11:             (G, s)= BugDetectionAndCorrection($\mathbb{G}$)
12:             gate G is buggy and s is the solution
13:     return Implementation is correct    ▷ if none of the constraints finds a non-zero remainder
```

gate. Note that the weight of each gates' output is computed by considering known weight of primary inputs and outputs, and traversing in both backward and forward directions while propagating the weights based on the approach outlined in [21]. ■

V. EXPERIMENTS

A. Experimental Setup

Incremental equivalence checking and debugging algorithms were implemented using a Java program and experiments were performed on an Intel core i7 Processor with 16 GB memory. We have tested our approach on post-synthesized gate-level integer arithmetic circuits that implement adders and multipliers. The post-synthesized integer arithmetic circuits are more difficult to verify and debug due to their optimized architectures and their carry chains. The designs were synthesized using Xilinx synthesis tool. We consider gate misplacement that changes the functionality of the design as our fault model. To illustrate that our debugging approach can fix errors form different stages of the design, we partitioned the implementation in four levels and several gates from these levels were randomly replaced with a faulty gate to create erroneous implementations. Inputs' constraints are generated automatically using a program. In order to generate the remainder, we have created three different threads: i) using no input constraints (as described in Section III which is similar to [6]), ii) using input constraints starting from the least significant input bits, and iii) using input constraints starting from the most significant bits. We have considered the fastest time for remainder generation among these three threads for the reported time in equivalence checking. In other words, when any one of these threads generates a remainder, the other two threads are terminated. We compared our equivalence checking and debugging results with state-of-the-art approaches [6], [9].

B. Results

Table III presents the results of our incremental equivalence checking and debugging approaches. The first column presents the type of the benchmarks which are either two-input ripple carry adders or two-input array multipliers. The second

TABLE III
THE RESULTS OF THE PROPOSED INCREMENTAL EQUIVALENCE AND DEBUGGING APPROACHES FOR INTEGER ARITHMETIC CIRCUITS. TO = TIMEOUT AFTER 5 HOURS; MO = MEMORY OUT OF 16 GB.

Type	Size	#Gates	Bug. Loc.	Equivalence checking (s)				Debugging (s)								
								[9]				Our Approach				Imp.
				Z3	[6]	Ours	Imp.	TG	BL	DC	Total	TG	BL	DC	Total	
Post-Syn Mult.	8x8	368	$0-1/4$	23.35	$\underline{0.02}$	0.02	1x	0.08	0	0.05	0.13	0.01	0	0.01	0.02	6.5x
			$1/4-2/4$	$\underline{22.83}$	45.31	0.2	114.15x	5.17	0.01	3.64	8.82	0.02	0	0.06	0.08	110.25x
			$2/4-3/4$	$\underline{22.59}$	TO	0.23	98.21x	_	_	_	_	0.12	0.01	0.44	0.58	$*$
			$3/4-4/4$	$\underline{23.13}$	TO	0.81	28.5x	_	_	_	_	0.04	0	0.23	0.27	$*$
	16x16	1.6K	$0-1/4$	107.1	$\underline{0.51}$	0.51	1x	0.34	0.01	0.69	1.04	0.02	0	0.03	0.05	20.8x
			$1/4-2/4$	$\underline{104.5}$	TO	1.42	73.6x	_	_	_	_	0.02	0.01	0.2	0.23	$*$
			$2/4-3/4$	$\underline{105.4}$	MO	1.58	66.7x	_	_	_	_	0.28	0.01	0.42	0.71	$*$
			$3/4-4/4$	$\underline{109.6}$	MO	0.31	353.54x	_	_	_	_	0.03	0.01	0.55	0.59	$*$
	32x32	7K	$0-1/4$	MO	$\underline{1.57}$	1.57	1x	0.75	0.1	5.27	6.12	0.08	0.01	0.92	1.01	6.05x
			$1/4-2/4$	MO	MO	1.33	$*$	_	_	_	_	0.06	0.01	4.74	4.81	$*$
			$2/4-3/4$	MO	MO	3.29	$*$	_	_	_	_	0.1	0.01	12.51	12.62	$*$
			$3/4-4/4$	MO	MO	1.58	$*$	_	_	_	_	0.05	0.06	6.36	6.47	$*$
	64x64	28K	$0-1/4$	MO	$\underline{16.43}$	16.43	1x	2.7	5	29.21	36.91	0.4	0.02	8.33	8.75	4.21x
			$1/4-2/4$	MO	MO	41.33	$*$	_	_	_	_	1.94	0.1	28.72	30.76	$*$
			$2/4-3/4$	MO	MO	80.25	$*$	_	_	_	_	1.74	0.1	346.16	348	$*$
			$3/4-4/4$	MO	MO	13.65	$*$	_	_	_	_	0.2	0.3	26.48	26.99	$*$
	128x128	132K	$0-1/4$	MO	$\underline{262.4}$	262.4	1x	4.7	28	379.4	412.1	3.5	0.5	89.15	93.15	4.42x
			$1/4-2/4$	MO	MO	1587.5	$*$	_	_	_	_	4.1	1.12	960.72	965.94	$*$
			$2/4-3/4$	MO	MO	1260.1	$*$	_	_	_	_	1.1	2.24	1219.8	1223.14	$*$
			$3/4-4/4$	MO	MO	303.83	$*$	_	_	_	_	0.35	3.5	75.35	79.29	$*$
Post-Syn Adder	64x64	573	$0-1/4$	51.36	$\underline{0.2}$	0.2	1x	0.69	0	0.31	0.82	0.12	0	0.02	0.14	5.85x
			$1/4-2/4$	$\underline{38.39}$	TO	3.18	12.07x	_	_	_	_	0.02	0	0.04	0.06	$*$
			$2/4-3/4$	$\underline{32.61}$	TO	3.33	9.79x	_	_	_	_	0.01	0	0.09	0.10	$*$
			$3/4-4/4$	$\underline{30.51}$	TO	1.17	26.07x	_	_	_	_	0.03	0	0.08	0.11	$*$
	128x128	1.2K	$0-1/4$	101.1	$\underline{1.16}$	1.16	1x	1.99	0.01	0.5	2.5	0.01	0	0.09	0.1	25x
			$1/4-2/4$	$\underline{115.8}$	TO	4.2	27.6x	_	_	_	_	0.01	0	0.12	0.13	$*$
			$2/4-3/4$	$\underline{116.9}$	MO	9.04	12.93x	_	_	_	_	0.02	0	0.27	0.29	$*$
			$3/4-4/4$	$\underline{115.9}$	MO	1.88	61.64x	_	_	_	_	0.05	0	0.34	0.39	$*$
	256x256	2.3K	$0-1/4$	158	$\underline{3.84}$	3.84	1x	3.35	0.1	1.51	4.96	0.11	0.01	0.07	0.19	26.10x
			$1/4-2/4$	$\underline{252.5}$	TO	93.4	2.7x	_	_	_	_	0.03	0.01	0.18	0.22	$*$
			$2/4-3/4$	$\underline{281.5}$	MO	40.62	6.93x	_	_	_	_	0.01	0.01	0.22	0.24	$*$
			$3/4-4/4$	$\underline{301.7}$	MO	20.35	14.82x	_	_	_	_	0.03	0.01	0.63	0.67	$*$

"*" indicates our approach works but existing method fails. "_" shows the cases when test generation, bug localization/correction cannot be done due to lack of the remainder.

and third columns indicate the input size ($firstOperand \times secondOperand$) and design size (number of gates), respectively. The fourth column shows the location of the bug. We partition the implementation into four levels, e.g. "$0-1/4$" refers to the gates closest to the primary inputs and "$3/4-4/4$" refers to the gates which are placed in the deeper stages of the design (closest to the primary outputs). The fifth column shows the equivalence checking result using Z3 SMT solver [24]. To use a SMT solver, we have converted polynomials (specification and implementation polynomials) into SMT solver format. The sixth column presents the equivalence checking (run-time in seconds) results of [6] for a faulty design. The seventh column indicates the time of our proposed incremental equivalence checking method which includes the time of the inputs' space partitioning as well as Algorithm 2. The eight column presents the improvement (Imp.) provided by our approach in comparison with the best result (indicated using underscore) of existing approach shown in fifth and sixth columns. In the table, "$*$" indicates that our approach performs well, whereas the existing approach [6] fails. The SMT solver tries to find a counterexample when implementation and specification are not equal. As it is shown, the SMT solver cannot find a counterexample for large designs. Moreover, there is no efficient and fully automatic debugging approach for fixing the bug using SMT solvers. It can be observed that when a bug exists on deeper stages of the design, this approach fails even for very small benchmarks. When the bug is close to the primary inputs, the incremental approach can take more time to finish since it goes through several iterations if they result in a zero remainders. However, when the bug is not near primary inputs, our approach not only makes the checking possible but also is faster by several orders-of-magnitude.

The next four columns show the time (in seconds) required for test generation (TG), bug localization (BL), debug (DC) and total (TG+BL+DC), respectively, using [9]. The subsequent four columns shows the same features using our proposed approach. The test generation time is dependent on the number of terms in the reminder. Since our approach generates more compact remainder, test generation time has improved significantly. If a non-zero remainder can be obtained using a smaller number of inputs' constraints, the remainder will be more compact. Since we use the constraints' order where most significant bits comes first, if the bug exists close to the primary outputs, the chance of obtaining a more compact remainder is more and test generation time is reduced. The results show that our approach requires less time to perform bug localization since the size of the generated remainder is small, and as a result, the number of directed tests are less. Moreover, the results show the effectiveness of incremental debugging approach based on required time for bug detection and correction compared to [9]. Based on the inputs order that we have considered, if a bug is located in middle stages of the design, the number of suspicious gates is increased, therefore, the time for bug correction and detection increases. In the table, "_" indicates that test generation, bug localization, and bug correction are not possible due to lack of the remainder. Finally, the last column presents the improvement (Imp.) provided by our incremental debugging approach. Clearly, our proposed approach can drastically reduce the overall debugging effort. Most importantly, it is able to debug hard-to-detect errors when existing state-of-the-art methods fail.

Table IV presents the equivalence checking time for correct

TABLE IV
THE EQUIVALENCE CHECKING TIME FOR CORRECT DESIGNS.

Benchmark	Size	Z3 SMT Solver	[5]	Our Approach	Imp. on Z3
Post-Syn. Multiplier	8x8	47.81	0.04	0.05	957.04x
	16x16	TO	0.11	0.16	*
	32x32	MO	0.42	0.43	*
	64x64	MO	2.50	2.68	*
	128x128	MO	19.25	19.77	*
Post-Syn. Adder	64x64	31.84	0.08	0.1	318.4x
	128x128	190.61	0.4	0.44	433.27x
	256x256	513.98	0.84	0.88	584.07

implementations of different designs. We have compared our proposed framework with Z3 SMT solver and [6]. As it can be observed, our method outperforms the Z3 and its performance is comparable with [6]. Table IV and V demonstrate that our approach performs well irrespective of whether the implementation is buggy or not.

Our experimental results highlight three important aspects of our debugging approach. First, using the inputs' constraints as well as the incremental debugging address the scalability issues of the existing arithmetic circuits' equivalence checking methods. Second, our incremental equivalence checking method enables more compact remainder generation that can improve the required time for test generation, bug localization and bug detection. Finally, the debugging approach automatically and efficiently detects and corrects unknown bugs regardless of its complexity and location in the design.

VI. Conclusion

In this paper, we presented an incremental equivalence checking and debugging framework for arithmetic circuits. The proposed approach made three important contributions. It partitions the primary inputs' space of the design in order to solve the verification and debug problems in the increasing order of the design complexity. Moreover, it developed an incremental equivalence checking algorithm to enable generation of compact remainders. Finally, the proposed incremental debugging enabled efficient bug detection and correction. Our experimental results demonstrated that our incremental verification framework is several orders-of-magnitude faster than existing state-of-the-art approaches.

VII. Acknowledgments

This work was partially supported by grants from National Science Foundation (CNS-1441667), Semiconductor Research Corporation (2014-TS-2554) and Cisco.

References

[1] U. Krautz, V. Paruthi, A. Arunagiri, S. Kumar, S. Pujar, and T. Babinsky, "Automatic verification of floating point units," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2014, pp. 1–6.

[2] R. E. Bryant and Y.-A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*. ACM, 1995, pp. 535–541.

[3] M. Haghbayan, B. Alizadeh, P. Behnam, and S. Safari, "Formal verification and debugging of array dividers with auto-correction mechanism," in *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*. IEEE, 2014, pp. 80–85.

[4] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 10, pp. 1606–1621, 2005.

[5] F. Farahmandi and B. Alizadeh, "Groebner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," *Microprocessors and Microsystems*, vol. 39, no. 2, pp. 83–96, 2015.

[6] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi, "Verification of gate-level arithmetic circuits by function extraction," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 52.

[7] T. Pruss, P. Kalla, and F. Enescu, "Equivalence verification of large galois field arithmetic circuits using word-level abstraction via gröbner bases," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[8] A. Sayed-Ahmed, D. Gro, M. Soeken, R. Drechsler *et al.*, "Formal verification of integer multipliers by combining gröbner basis with logic reduction," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1048–1053.

[9] 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.

[10] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G.-M. Greuel, "An algebraic approach for proving data correctness in arithmetic data paths," in *International Conference on Computer Aided Verification*. Springer, 2008, pp. 473–486.

[11] S. Safarpour and A. Veneris, "Abstraction and refinement techniques in automated design debugging," in *Seventh International Workshop on Microprocessor Test and Verification (MTV'06)*. IEEE, 2006, pp. 88–93.

[12] B. Keng and A. Veneris, "Path-directed abstraction and refinement for sat-based design debugging," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 10, pp. 1609–1622, 2013.

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

[14] A. Sülflow, G. Fey, and R. Drechsler, "Experimental studies on smt-based debugging," in *IEEE Workshop on RTL and High Level Testing*, 2008, pp. 93–98.

[15] T. Matsumoto, S. Ono, and M. Fujita, "An efficient method to localize and correct bugs in high-level designs using counterexamples and potential dependence," in *VLSI and System-on-Chip, 2012 (VLSI-SoC), IEEE/IFIP 20th International Conference on*. IEEE, 2012, pp. 291–294.

[16] B. Keng, S. Safarpour, and A. Veneris, "Bounded model debugging," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 11, pp. 1790–1803, 2010.

[17] M. K. Ganai and A. Gupta, "Efficient bmc for multi-clock systems with clocked specifications," in *2007 Asia and South Pacific Design Automation Conference*. IEEE, 2007, pp. 310–315.

[18] J. A. Kumar, S. N. Ahmadyan, and S. Vasudevan, "Efficient statistical model checking of hardware circuits with multiple failure regions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 6, pp. 945–958, 2014.

[19] S. Vasudevan, J. A. Abraham, V. Viswanath, and J. Tu, "Automatic decomposition for sequential equivalence checking of system level and rtl descriptions," in *Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings.* IEEE Computer Society, 2006, pp. 71–80.

[20] Y. T. Chang and K. T. T. Cheng, "Induction-based gate-level verification of multipliers," in *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. IEEE Press, 2001, pp. 190–193.

[21] S. Ghandali, C. Yu, D. Liu, W. Brown, and M. Ciesielski, "Logic debugging of arithmetic circuits," in *2015 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2015, pp. 113–118.

[22] X. Sun, P. Kalla, T. Pruss, and F. Enescu, "Formal verification of sequential galois field arithmetic circuits using algebraic geometry," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 1623–1628.

[23] B. Buchberger, "A criterion for detecting unnecessary reductions in the construction of gröbner-bases," in *Symbolic and algebraic computation*. Springer, 1979, pp. 3–21.

[24] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.