Logic Debugging of Arithmetic Circuits

Samaneh Ghandali, Cunxi Yu, Duo Liu, Walter Brown, Maciej Ciesielski
University of Massachusetts, Amherst, USA
{samaneh, ycunxi, duo, webrown, ciesiel}@umass.edu

Abstract— **This paper presents a novel diagnosis and logic debugging method for gate-level arithmetic circuits. It detects logic bugs in a synthesized circuit caused by using a wrong gate ("gate replacement" error), which change the functionality of the circuit. The method is based on modeling the circuit in an algebraic domain and computing its algebraic "signature". The location and type of the bug is determined by comparing signatures computed in both directions, using forward (PI to PO) and backward (PO to PI) rewriting. It will also perform automatic correction for the detected bugs. The approach is demonstrated and tested on a set of integer combinational arithmetic circuits.**

Keywords— Formal verification; Logic debugging; Arithmetic circuits.

I. INTRODUCTION

As today's VLSI designs grow in complexity and size, design errors become more frequent and difficult to track [1]. The process of verifying the functional correctness of a design, determining the source of potential errors and correcting those errors, can take up to 70% of the overall design time [2]. Recent developments have automated most of the verification tasks, but debugging, i.e., error localization and correction, still remains a resource intensive, manually conducted process [4]. Efficient automated debugging techniques are necessary to complement and enhance the verification techniques.

Traditional automated debugging solutions for hardware designs are based on simulation, critical path tracing [5], BDDs and *BMDs [6]. Recent automated debugging methods tend to rely on SAT solvers. In [7] error detection is facilitated by adding corrector models to the circuit implementation and mapping it into a Boolean formula in CNF. By solving the resulting SAT problem, a set of suspect error locations are obtained. This approach, however, is restricted by the performance and capacity of the available SAT solvers. Other techniques, such as those based on Quantified Boolean Formula (QBF) [8], abstraction and refinement in error localization [9], and maximum satisfiability [10], [11], are used to improve SAT-based debugging method. However, the performance of these methods and their capability to handle large designs remain limited by SAT. In order to reduce the number of SAT solver calls, the concept of reverse dominators was introduced in [12] to allow for early pruning of non-solution areas of the problem search space. FPGA-based debugging methodology, proposed in [13], [14], [15], locally modifies the circuit structure.

The diagnosis problem for sequential circuits is typically structured as bounded model checking (BMC) [16], [17] and formulated as a SAT problem. In [17], resynthesis method guided by counterexamples performs gate-level circuit repair, based on error traces composed of input vectors and output responses. The method described in [18],[19] introduces an abstraction and refinement algorithm for design debugging built upon a time-windowing framework to manage excessive error trace lengths. Non-modeled portions of the trace are approximated using a path directed abstraction that represents structural circuit paths. Due to the inherent iterative nature of the algorithm, performance remains the crucial issue in this work.

A debugging technique applicable to divider circuits is proposed in [20]. It is based on a "reverse-engineering" mechanism of extracting a high level arithmetic model, called Functional Bit Level Adder (FBLA), which may be difficult to obtain in synthesized circuits. Furthermore, these methods can only reason about the correctness of the quotient part of the result using iterative subtraction model, but not about the entire divider circuit. In [21] verification of RTL code to optimize assertion coverage is proposed. Their algorithm can be used to isolate only those statements that are covered by an assertion as the most likely location of the bug. However, the problem of generating assertions remains open.

In this paper, we introduce a novel diagnosis and logic debugging method for gate-level arithmetic circuits. The proposed method is part of the functional verification approach proposed in [3]. It detects the logical bugs, caused by using a wrong gate ("gate replacement" bug) or inversion of an internal signal, that change the functionality of the circuit. It will also perform automatic correction for the detected bugs. The approach assumes a "single-gate" replacement error, caused by using a wrong gate, but it can correct multiple independent bugs. It consists of three phases: 1) the circuit is scanned forward from the primary inputs (PI) to primary outputs (PO) and an algebraic expression (signature) is derived for each cut (a set of signals that separate PI from PO); 2) the circuit is scanned backward from PO to PI and an algebraic signature is generated for each cut; 3) the difference between the two expressions, Δ_i, at each cut is then computed. A non-zero Δ_i for a given cut indicates inconsistency between the two expressions, showing that there is a bug located at this cut. The value of Δ_i is then analyzed to determine the source of the bug and to correct it. Under certain conditions several bugs in the same cut can be corrected simultaneously.

The rest of this paper is organized as follows. Section II describes preliminaries. Section III explains in detail the proposed diagnosis and debugging method. Section IV presents experimental results and Section V provides summary and conclusions.

II. Preliminaries

We follow the arithmetic verification approach proposed in [3], with the circuit modeled as a network of basic logic gates (AND, OR, XOR, INV, etc.). Each gate is represented as a pseudo-Boolean polynomial *poly[X]*, with Boolean variables $X = \{x_1, ..., x_n\}$ and integer coefficients from Z_2^n. The following equations summarize algebraic representation of basic Boolean operators:

$$\neg a = 1 - a$$
$$a \wedge b = a\, b$$
$$a \vee b = a + b - a\, b \qquad (1)$$
$$a \oplus b = a + b - 2\, a\, b$$

Definition 1 *(Input Signature)*: The input signature, Sig_{in}, is a polynomial in primary input variables that uniquely represents an integer function computed by the circuit, i.e., its specification. For example, an *n*-bit binary adder with inputs $\{a_0,...,a_{n-1},b_0,...,b_{n-1}\}$, is described by $Sig_{in} = \sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i$. The input signature of a 2-bit signed multiplier is $Sig_{in} = (-2a_1+a_0)(-2b_1+b_0) = 4a_1b_1-2a_0b_1-2a_1b_0+a_0b_0$, etc. The integer *coefficients* (weights) associated with the circuit signals are uniquely determined by the intended circuit function (specification). For example, in an adder, the coefficients of the primary inputs at bit position *i* are $c(a_i) = c(b_i) = 2^i$.

Definition 2 *(Output Signature):* the output signature, Sig_{out}, of the circuit is defined as a polynomial in the primary output signals. Such a polynomial is uniquely determined by an *n*-bit encoding of the output provided by the designer. For example, the output signature of the 2-bit signed multiplier is $-8z_3+4z_2+2z_1+z_0$. In general, an output signature of an unsigned arithmetic circuit with *n* output bits is represented as a linear polynomial, $Sig_{out} = \sum_{i=0}^{n-1} 2^i z_i$. The coefficients of the primary outputs are also unique, defined by the known output encoding.

Definition 3 *(Cut Signature)*: The *cut* is a set of signals that separates PI from PO. The *signature* of a cut is a polynomial expression in signal variables of the cut that represents the integer number computed by the circuit.

The selection of the cuts is an important issue in this approach, as it affects the efficiency of finding the bugs. In the worst case, two cuts may only differ by a single gate. For illustration purpose we assume that the cuts are determined by the topological ordering of signals w.r.t. PI, but other choices exist (determining the best set is part of the future work).

Example 1: Figure 1 shows a two-bit adder, with $Sig_{in} = 2a_1+2b_1+a_0+b_0$, $Sig_{out} = 4r_2+2r_1+r_0$, and "topological" cuts, labeled $f_0,...,f_3$. The meaning of Δ_i in the figure will be explained in Section III.C. This circuit will be used as a running example in the paper.

As shown in [3], in a bug-free arithmetic circuit, the expressions for any two cuts, although expressed by different polynomials, always evaluate to the same value, i.e., $f(cut_i) = f(cut_j)$, for any $\{i, j\}$. This fundamental property of the arithmetic circuit serves as basis of the proposed diagnostics and debugging approach.

III. Bug Identification

Our debugging method consists of: computing cut signatures by forward rewriting, backward rewriting, and comparing the pairs of signatures for each cut to identify and fix the bugs.

A. Forward (PI-PO) rewriting

The forward (PI-PO) rewriting starts by dividing the initial polynomial, Sig_{in}, by the polynomials describing the logic gates connected to the PI signals. The goal is to replace the input variables associated with the PI gates with an expression involving the corresponding gate outputs. This produces an expression in the new set of variables, moving away from PI. While in principle this can be done one gate at a time, one can eliminate several gates at once to speed up the process. To do this division efficiently, knowledge of the signal *coefficients* (weights) is needed. We explain how to calculate the required coefficients of the newly introduced variables using the structures shown in Figure 2.

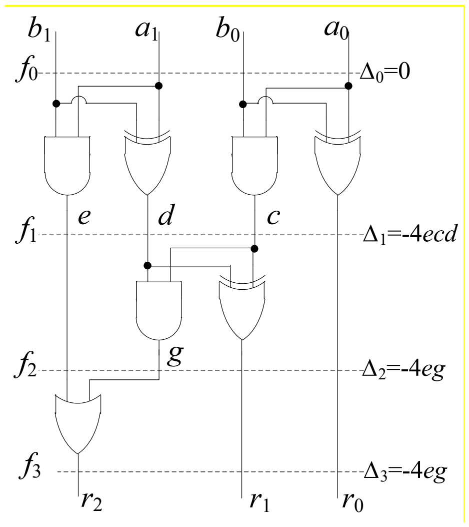


Fig. 1. Two-bit adder circuit with topological cuts.

Figure 2(a) shows a half-adder (HA) circuit, consisting of a pair (XOR, AND) with common variables. Using the notation in the figure and the algebraic representation of the logic gates given in Eq.(1), the computation of the output coefficients of the half-adder circuit with inputs (*a,b*) and outputs (*m,n*) is performed as follows: $Sig_{in}(HA) = c_1a+c_1b$

$Sig_{out}(HA) = c_2m+ c_3n = c_2(a+b-2ab) + c_3ab$

Since $Sig_{in}(HA) = Sig_{out}(HA)$, we have:

$c_1a+c_1b - c_2(a+b-2ab) - c_3ab = 0.$

By regrouping the variables as follows

$a(c_1 - c_2) + b(c_1 - c_2) + ab(2c_2 - c_3) = 0$

and solving the above equation for c_2 and c_3, we obtain:

$c_2 = c_1$ and $c_3 = 2c_1$.

The second structure shown in Fig. 2(b) consists of an XOR gate and an OR gate. Using similar approach, we obtain: $c_2 = -c_1$ and $c_3 = 2c_1$. Similarly, the structure in Fig. 2(c), consisting of an OR and an AND gate, produces the coefficients: $c_2 = c_1$ and $c_3 = c_1$.

Coefficients of the individual gates can be derived similarly. It can be shown that the inputs to an OR or XOR gate must have the same coefficients, $c_1 = c_2$, otherwise the algebraic equation for this gate will not be satisfied; the coefficient c_3 of the output is equal to those of the input. In contrast, the input coefficients c_1, c_2 of an AND gate can be different, and the output coefficient $c_3 = c_1 \cdot c_2$.

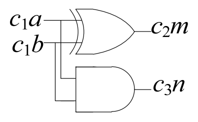
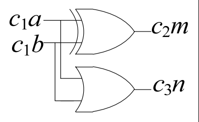
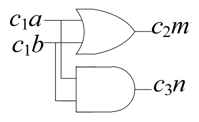
 $c_2 = c_1$ $c_3 = 2c_1$	 $c_2 = -c_1$ $c_3 = 2c_1$	 $c_2 = c_1$ $c_3 = c_1$
(a)	(b)	(c)

Fig. 2. Calculation of signal coefficients.

Computing cut signatures: The pseudo code of the algorithm for forward rewriting equation for each cut is shown in **Algorithm 1**. The input to this algorithm is the circuit with its input signature, and its output is a set of cut equations. Equation of the first cut is the same as the specification, i.e., $f(cut_0) = Sig_{in}$. Obtaining cut_i from cut_{i-1} works as follows. Let x_j be an output signal of gate g_j in cut_i and $poly(g_j)$ be the polynomial expression describing its logic function (*c.f.* Eq.(1)), so that $x_j = poly(g_j)$. For each gate g_j in cut_i we add variable x_j to the current cut and subtract the algebraic expression $poly(g_j)$ representing this gate, without changing the arithmetic function of the cut. Equation (2) shows the computation of $f(cut_i)$ from $f(cut_{i-1})$.

$$f\left(cut_i\right) = f\left(cut_{i-1}\right) + \sum_{j=1}^{gates\in cut_i} c_j\left(x_j - poly\left(g_j\right)\right) \qquad (2)$$

Here, i is the index of the cut, and c_j is the coefficient of gate j of cut_i. Lines 3-7 of Algorithm 1 describe the computation of each cut. For simplicity, we write f_i instead of $f(cut_i)$. Each f_i is initialized with f_{i-1}; using the structure in Fig. 2, for each gate j of cut_i, the gate coefficient is calculated so as to eliminate the gate inputs (line 6); finally, equation of cut_i is computed (line 7), based on Eq. (2).

```
Algorithm Forward (PI-PO) rewriting ($Sig_{in}$, $Circuit$)
1 Compute all cuts of the Circuit;
2 $f_0 = Sig_{in}$;
3 for $i$ = 1 to # of cuts
4     $f_i = f_{i-1}$;
5     for $j$=1 to #of gates of cut$_i$
6         $c_j$ = compute coefficient of $g_j$;
7         $f_i = f_i + c_j(x_j - poly(g_j))$;
```
Algorithm 1. Forward (PI-PO) rewriting

Example 2: Consider again the circuit in Fig. 1. By applying the forward (PI-PO) rewriting algorithm, we obtain the following equations for each cut:

$$f_0 = 2b_1 + 2a_1 + b_0 + a_0$$
$$f_1 = 2b_1 + 2a_1 + b_0 + a_0 + 4(e - a_1b_1) + 2(d - (a_1 + b_1$$
$$- 2a_1b)) + 2(c - a_0b_0) + (r_0 - (a_0 + b_0 - 2a_0b_0))$$
$$= 4e + 2d + 2c + r_0$$
$$f_2 = 4e + 2d + 2c + r_0 + 4(g - dc) + 2(r_1 - (d + c - 2dc))$$
$$= 4e + 4g + 2r_1 + r_0$$
$$f_3 = 4e + 4g + 2r_1 + r_0 + 4(r_2 - (e + g - eg))$$
$$= 4r_2 + 2r_1 + r_0 + 4eg$$

Note that such computed f_3 is different than the expected output signature, $4r_2+2r_1+r_0$. Specifically, it contains the term $4eg$, associated with variables of cut_2. We call such a term a *Residual Expression (RE)*. In a correct circuit, *RE* should be zero. This can be proved by a straightforward rewriting of $4eg$ up to the PI variables:

$$4eg = 4(a_1b_1)(dc) = 4(a_1b_1)(a_1+b_1-2a_1b_1)(a_0b_0) = 0$$

Then, $f_3 = 4r_2+2r_1+r_0$, indicating that the circuit is correct. The reason for the existence of a residual expression in a correct circuit is that the polynomial division used by forward rewriting does not take into account the Boolean nature of the circuit signals. To avoid *RE* one would need to divide the polynomials by a set of polynomials $<x^2-x>$, called *ideals*, for each signal x in the circuit, to guarantee that $x=0,1$. This method, often employed by symbolic algebra approach, is too costly and inefficient for this work.

B. Backward (PO-PI) rewriting

The backward (PO-PI) rewriting is conceptually simpler, basically a reversed symbolic simulation. Starting at the PO with Sig_{out}, it creates a new cut signature by replacing an output signal x_j of gate g_j with its corresponding algebraic expression: $x_j \rightarrow poly(g_j)$. Here the signal coefficients are known, provided by the binary encoding of the PO signals.

Example 3: The cut expressions for the two-bit adder in Fig. 1 computed in PO-PI fashion are as follows:

$$f_3 = 4r_2 + 2r_1 + r_0$$
$$f_2 = 4(g + e - eg) + 2r_1 + r_0 = 4g + 4e - 4eg + 2r_1 + r_0$$
$$f_1 = 4e + 4(\mathrm{cd}) - 4\mathrm{e}(cd) + 2(\mathrm{c} + d - 2cd) + r_0$$
$$= 4e + 2d + 2c + r_0 - 4ecd$$
$$f_0 = 4e + 2d + 2c + r_0 - 4ecd$$
$$= 4(\mathrm{a}_1 b_1) + 2(a_1 + b_1 - 2\,\mathrm{a}_1 b_1) + 2(\mathrm{a}_0 b_0) + (a_0 + b_0$$
$$- 2\,\mathrm{a}_0 b_0) - 4(\mathrm{a}_1 b_1)(\mathrm{a}_0 b_0)(a_1 + b_1 - 2\,\mathrm{a}_1 b_1)$$
$$= 2a_1 + 2b_1 + a_0 + b_0$$

The computed signature at the PI matches the expected specification, Sig_{in}, so the circuit is correct. Note that the backward rewriting will never produce a residual expression. This is because the algebraic model (1) of Boolean gates correctly represents the binary value of the gate signal. This convenience comes at a cost of a potentially exponential explosion of the signature size during backward rewriting.

C. Computing the signature difference (Δ_i)

At this point, a pair of expressions is generated for each cut of the circuit: one computed by the forward and the other by the backward rewriting. The difference Δ_i between the two expressions is defined as follows:

$$\Delta_i = f_{i,BACK} - f_{i,FOR}; \quad 0 \le i \le cuts \qquad (3)$$

Here, i is the index of cut_i, $f_{i,BACK}$ is the signature of cut_i in the PO-PI direction, and $f_{i,FOR}$ is the signature of cut_i computed by the PI-PO rewriting. If the circuit contains no bug, the value of Δ_i at each cut is equal to zero; otherwise, the circuit contains a bug. The expression of Δ_i will be used to identify and to correct the bug.

Example 4: Consider the adder circuit in Examples 2 and 3 again. The values of parameter Δ_i for each cut of the circuit are shown in Fig. 1. As can be seen, Δ_3, Δ_2 and Δ_1 are non-zero polynomials. However, as explained in Example 3, *4eg* and *4ecd* are zero functions (expressions that evaluate to zero), so Δ_3, Δ_2 and Δ_1 also reduce to zero.

D. The Debugging Algorithm

In this phase, the circuit is analyzed and verified against the given specification to either confirm its correctness or to find and locate the bug. In principle, the circuit is correct (satisfies its specification) if the signature obtained by backward rewriting matches the given input signature (specification). Alternatively, the signature obtained by the forward rewriting should match the given output signature, provided that the residual expression *RE* generated during this rewriting is proven to be zero (as explained earlier, this can be done by a local backward rewriting of the RE expression up to PI). In this paper we consider a particular type of a bug, namely *gate replacement*, i.e., using a wrong gate in the circuit. In practice, in the presence of a bug, the size of the computed signature may become prohibitively large, and the goal is to locate the cut at which the bug (faulty gate) resides. If the bug is located at some cut_i, then the value of Δ_i for this cut will be nonzero. This is illustrated by the following example.

Example 5: Let us intentionally insert a bug into the two-bit adder circuit in Fig. 1, by replacing the AND gate with inputs (*c,d*) with an OR gate. The resulting buggy circuit is shown in Fig. 3. The cut equations for this circuit are as follows.

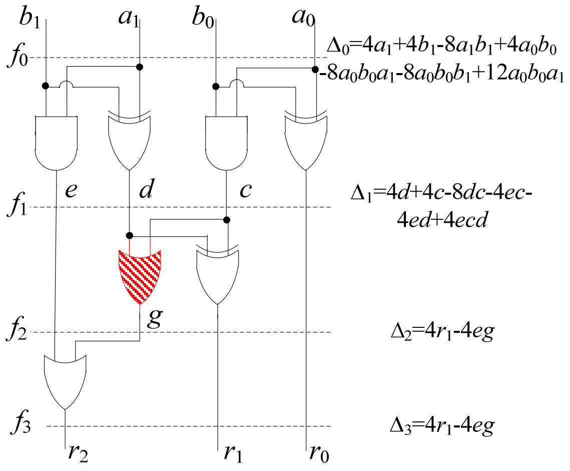
Forward (PI-PO) rewriting:

$$f_0 = 2b_1 + 2a_1 + b_0 + a_0$$
$$f_1 = 4e + 2d + 2c + r_0$$
$$f_2 = 4e + 4g - 2r_1 + r_0$$
$$f_3 = 4r_2 - 2r_1 + r_0 + 4eg$$

Backward (PO-PI) rewriting:

$$f_3 = 4r_2 + 2r_1 + r_0$$
$$f_2 = 4e + 4g - 4eg + 2r_1 + r_0$$
$$f_1 = 4e + 6d + 6c - 8cd - 4ec - 4ed + 4ecd + r_0$$
$$f_0 = 6\mathrm{a}_1 + 6b_1 - 8\mathrm{a}_1 b_1 + \mathrm{a}_0 + b_0 + 4\mathrm{a}_0 b_0 - 8\mathrm{a}_0 b_0 \mathrm{a}_{11}$$
$$- 8\mathrm{a}_0 b_0 b + 12\mathrm{a}_0 b_0 \mathrm{a}_1 b_1$$

The values of parameter Δ_i for each cut of the buggy circuit are calculated using Eq. (2) and shown in Fig. 3. The type and the location of the bug can be obtained by assessing the value of Δ_i for each cut. Assume initially that each cut has only a single bug (the constraint to be removed later). **Table I** shows the difference in the signatures between the cut with the correct gate and the cut with the wrong gate. As an example, consider the entry (*a+b-2ab)* in the 1st row (AND) and 2nd column (OR) of the table. It reflects the difference between the correct AND gate (*ab*) and the wrong OR gate (*a+b-ab).* That is, if in a given cut, an AND gate is replaced with an OR gate, then Δ_i = *(a+b-ab)-ab =a+b-2ab*, where *a*, *b* are the gate inputs. Conversely, if for some cut computed by the algorithm, Δ_i =*a+b-2ab,* this means that it contains an OR gate while it should contain an AND.



b_1
a_1
b_0
a_0
f_0
Δ_0=4a_1+4b_1-8a_1b_1+4a_0b_0 -8$a_0b_0a_1$-8$a_0b_0b_1$+12$a_0b_0a_1$
e
d
c
f_1
Δ_1=4d+4c-8dc-4ec-4ed+4ecd
g
f_2
Δ_2=4r_1-4eg
f_3
Δ_3=4r_1-4eg
r_2
r_1
r_0

Fig. 3. Two-bit adder with a bug: OR instead of AND

The remaining entries in the table give expressions for different bugs for a single-gate replacement. To detect the location of the bug in a given cut we need to check if Δ_i contains any of the expressions in Table I (replaced by the appropriate variable names). If this is the case, the location of the bug is detected and can be corrected as specified in the table. Otherwise, either the bug originates at a different cut, or it has a different nature, not considered in this model.

TABLE I. EXPRESSIONS CAUSED BY GATE REPLACEMENT ERROR

Buggy / Correct	AND	OR	XOR
AND		*a+b-2ab*	*a+b-3ab*
OR	*-a-b+2ab*		*-ab*
XOR	*-a-b+3ab*	*ab*	

Table I can be readily extended to other types of logic gates, such as the complex And-Or-Invert gates used in standard cell implementations.

The pseudo code of the Debugging Algorithm is shown in **Algorithm 2**. The equation of each cut is first computed in a PI-PO direction (line 1) and then by PO-PI rewriting (line 2). Then, Δ_i is calculated for each cut. If Δ_0 =0, the circuit is correct, otherwise Δ_i of other cuts are assessed and their expressions are checked against those in Table I (lines 3-8). The detected bugs at each cut are stored in the set *Bug-list,* which consists of the potential bug locations.

The expression of each Δ_i is then modified by adding to it expression ($f_{i,FOR}$ - $f_{i+1,FOR}$), where $f_{i,FOR}$ and $f_{i+1,FOR}$ are the expressions obtained by *forward* rewriting of cut_i and cut_{i+1}, respectively (lines 9-11). This is done to account for the residual expression generated during the forward rewriting between cut_i and cut_{i+1}, since during the forward rewriting, the residual expression that is based on the variables of cut_i actually appears in cut_{i+1}. Note that only those terms of expression ($f_{i,FOR}$ - $f_{i+1,FOR}$) that belong to cut_i are added to Δ_i. The new Δ_i is examined again to check for the buggy expressions (lines 12-13).

```
Algorithm Debugging
1  Scan forward (PI-PO) and write equation for each $cut_i$: $f_{i,FOR}$;
2  Scan backward (PO-PI) and write equation for each $cut_i$: $f_{i,BACK}$;
3  For each $cut_i$
4      compute $\Delta_i = f_{i,BACK} - f_{i,FOR}$;
5      If ($\Delta_0 == 0$)
6          Return "No Bug";
7      If $\Delta_i$ contains expression in Table I
8          add detected bug to Bug_list;
9  If (bug_list is empty)
10     For each $cut_i$
11         $\Delta_i = \Delta_i + (f_{i,FOR} - f_{i+1,FOR})$;
12         If $\Delta_i$ contains an expression in Table I
13             add detected bug to Bug_list;
14 If (Bug_list is empty)
15     Return "Bug cannot be detected by our method";
16 For each member of Bug_list
17     Correct the circuit;
18     Compute $\Delta_0$;
19     If ($\Delta_0 == 0$)
20         Return "Bug is detected and corrected";
```
Algorithm 2. Pseudo code of the Debugging Algorithm

The *Bug-list* shows all potential bug locations. If after modifying the Δ_i expressions the *Bug-list* remains empty, this means that our debugging algorithm cannot detect the bug (lines 14-15). To detect the exact location of the bug and correct the circuit, the first bug of the *Bug-list* is replaced with the corresponding correct gate from Table I. Then the circuit is verified by re-computing Δ_0. If $\Delta_0 \neq 0$, we consider the next bug from the list and to correct it in the buggy circuit (lines 19-20). This process is repeated until the circuit becomes correct, i.e., until $\Delta_0 = 0$.

Example 6: Continuing with Example 5, we compute Δ_i for all the cuts, trying to match their variables with one of the expressions in Table I. Recall that only the signals from the given cut must be used in the matching. At cut_3, with signals $\{r_2,r_1,r_0\}$, $\Delta_3 = 4r_1 - 4eg$; but $4r_1$ does not match any of the expressions in the table. At cut_2, with signals $\{e,g,r_1,r_0\}$, we have $\Delta_2 = 4r_1 - 4eg$. Here we find that $-4eg$ matches an expression in the table ($-ab$), at the OR/XOR entry of the table. However, cut_2 does not have any XOR gate, so this cannot be the source of the bug. At cut_1, with signals $\{e,d,c,r_0\}$, we have $\Delta_1 = 4d+4c-8dc-4ec-4ed+4ecd$. Here $4d+4c-8dc$ matches the buggy expression ($a+b-2ab$) with coefficient 4, indicating that an OR gate was used instead of an AND gate. As shown in Fig. 4, there is an OR gate at cut_1 with inputs $\{c, d\}$ so a bug is detected. This bug is then corrected by replacing the OR with an AND. The other terms ($-4ec$, $-4ed$) match the expression in the table (OR-XOR gate replacement). But at cut_1 there are no gates with inputs $\{e, c\}$ or $\{e, d\}$; hence no additional bug is reported. In a similar fashion, we can verify that there are no bugs in cut_0. Therefore, the only bug detected is at cut_1, caused by the OR gate in place of an AND. To correct the circuit, we just need to replace the OR gate with inputs (c, d) with an AND gate with the same inputs.

IV. Experimental Results

The algorithm has been implemented in C#. The experiments were conducted on a PC with Intel 1.80-GHz Core i7 processor and 6 GB of memory under Windows 8. We tested gate-level circuits of arithmetic functions: $F_1 = A+B$ and $F_2 = A\times B$, with bit-widths ranging from 32 to 128 bits. Several bugs (erroneous gates) were inserted in the middle of each circuit. Note that the bugs located near PI are easiest to detect (there is no residual expression) and the bugs inserted near POs are most difficult to detect (the signature with backward rewriting may explode in size). Table II and III show the results for the two circuits containing multiple bugs.

TABLE II. Debugging of $F_1 = A + B$ with multiple bugs

Bit-width	# Gates	# Bugs	Memory	CPU time (sec)
32	414	1	4.2 MB	1.46
		3	4.2 MB	1.56
		5	4.2MB	1.61
64	810	1	4.6 MB	3.40
		3	4.6 MB	3.51
		5	4.6 MB	3.58
128	1,662	1	6.3 MB	6.70
		3	6.3 MB	6.92
		5	6.3 MB	6.98

Our method can detect and correct every inserted bug in all the instances of the tested circuits in a reasonable time. The table demonstrates a linear CPU time dependence in the number of bugs (for a small number of bugs performed in this experiment).

TABLE III. Debugging of $F_2 = A\times B$ with multiple bugs

Bit-width	# Gates	# Bugs	Memory	CPU Time (sec)
32	8,062	1	8.9 MB	18.32
		3	9.5 MB	23.50
		5	11.2 MB	30.78
64	32,512	1	85 MB	184.50
		3	91 MB	189.43
		5	95 MB	194.45
128	131,072	1	122 MB	1927.40
		3	131 MB	2027.56
		4	143 MB	2136.86

V. Conclusions

The goal of this work was to provide a proof of concept for identifying and correcting bugs caused by gate replacement in gate-level arithmetic circuits. Despite its preliminary nature, the initial results demonstrate the validity and potential of the proposed approach for solving practical problems. The limitation of the method is generation of cuts as a means to locate the bugs. Determining the best set of cuts to improve the efficiency of the method is the major goal of our future work. One possibility is to adopt a "binary search" approach, by sampling the circuit with selected cuts and checking if their signature is correct. This may need to be supported by random backward simulation to help qualify the cut as correct or incorrect. The next cut in the sequence will then be placed half-way between the faulty one and the PO and the search for bugs will continue in this area in a similar fashion. The method is applicable to locating multiple bugs associated as long as they correspond to disjoint sets of variables.

ACKNOWLEDGMENT

This work was supported by a grant from the National Science Foundation, award No. CCF-1319496.

REFERENCES

[1] M.F. Ali, S. Safarpour, A. Veneris, M.S. Abadir, "Post-Verification Debugging of Hierarchical Designs," IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 871-876, 2005.

[2] Y. Chen, S. Safarpour, A. Veneris, J.M. Silva, "Spatial and temporal design debug using partial MaxSAT," Great Lakes symposium on VLSI (GLVLSI), pp. 345-350, 2009.

[3] M. Ciesielski, C. Yu, W. Brown, D. Liu, "Verification of Gate-level Arithmetic Circuits by Function Extraction," ACM Design Automation Conference (DAC-2015), 2015.

[4] Y. Yang, S. Sinha, A. Veneris, R. Brayton, "Automating Logic Rectification by Approximate SPFDs," Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 402-407, 2007.

[5] M. Abramovici, P.R. Menon, D.T. Miller, "Critical path tracing-an alternative to fault simulation," Design Automation Conference (DAC), 1983, pp. 214–220.

[6] R.E. Bryant, Y-A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams," Design Automation Conference (DAC), pp. 535–541, 1995.

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

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

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

[10] S. Safarpour, H. Mangassarian, A. Veneris, M.H. Liffiton, K.A. Sakallah, "Improved design debugging using maximum satisfiability," Formal Methods in Computer Aided Design (FMCAD), pp. 13–19, 2007.

[11] Y. Chen, S. Safarpour, J. Marques-Silva, A. Veneris, "Automated design debugging with maximum satisfiability," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 29, no. 11, pp. 1804–1817, 2010.

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

[13] M. Kubo, M. Fujita, "Debug methodology for arithmetic circuits on FPGAs," IEEE International Conference on Field-Programmable Technology (FPT), pp. 236–242, 2002.

[14] S. Yang, H. Shim, W. Yang, C-M Kyung, "A new RTL debugging methodology in FPGA-based verification platform," Asia-Pacific Conference on Advanced System Integrated Circuits, pp. 180–183, 2004.

[15] W. Li, Z.J. Song, A.W. Ruan, C.Q. Li, D.S. Yu, "A two-mode debugging system for vlsi designs using xilinx FPGA," International Conference on Computational Problem-Solving (ICCP), pp. 84–88, 2012.

[16] M.K. Ganai, A. Gupta, "Efficient BMC for multi-clock systems with clocked specifications," Asia and South Pacific Design Automation Conf. (ASP-DAC), pp. 310–315, 2007.

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

[18] B. Keng and A. Veneris, "Path directed abstraction and refinement in SAT-based design debugging," Design Automation Conference (DAC), 2012, pp. 947–954.

[19] B. Keng, A.G. Veneris, "Path-Directed Abstraction and Refinement for SAT-Based Design Debugging," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 32, n. 1o, pp. 1609-1622, 2013.

[20] M.H. Haghbayan, B. Alizadeh, A.M. Rahmani, P. Liljeberg, H. Tenhunen, "Automated formal approach for debugging dividers using dynamic specification," IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pp. 264–269, 2014.

[21] V. Athavale, S. Ma, S. Hertz, S. Vasudevan, "Code coverage of assertions using rtl source code analysis," Design Automation Conference (DAC), pp. 1–6, 2014.