Counterexample for Farimah's Debug Approach

Vikas Rao
Department of Electrical and Computer Eng.
University of Utah
Vikas.k.rao@utah.edu

November 8, 2017

1 Introduction

The most recent approach which deals with automatic debugging mechanism [1] is limited to textbook structure of arithmetic circuits. The coefficient computation in [1] borrowed from [2] is incomplete as it completely relies on the half adder structure and doesn't talk about the ambiguities in coefficient weight calculations when the gate structure differs from the given topology. The approach also fails to arrive at a conclusive solution when the circuit is tweaked with some redundant gates as shown in following sections.

2 Debug Approach

In the paper described [1], the approach for automatic debugging of arithmetic circuits is classified into following sub-problems: The first step is to do an equivalence check of the given circuit C against a golden specification f and confirm if at all the circuit is buggy, let us call this step as the remainder generation. Once we have a remainder, we need to identify the test patterns which excite the bug in the design, let us call this step as test generation. Once we identify the test vectors, we need to simulate these vectors to identify all the output bits which differ against the specification outputs (this helps in pruning the potential faulty gate list), let us call this step as fault pruning. The next step is to localize the bug, meaning identify the exact bug location, followed by which we need to find the correct fix such that it conforms to the specification.

We can have either one faulty gate or multiple faulty gates. For simplicity, we shall take a single faulty gate model, i.e.,only one gate in the design has been incorrectly replaced, for example an AND gate replaced with an XOR/OR gate. Based on this model we shall analyze and discuss the debug approach.

3 Preliminaries

To perform verification, the algebraic model of the implementation is used, wherein each gate is modeled as a polynomial with integer coefficients, and variables from $\mathbb{Z}_2$ are used. The polynomial computation for the respective gates are as shown below -

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

The construction of remainder to match the pattern considers the coefficients carried by the gate inputs as well, which is derived from [2] and summarized in figure 1. Coefficients of the individual gates can be derived similarly. It can be shown that the inputs to an XOR or OR gate must have the same coefficients($c_1 = c_2$), otherwise the algebraic equation for this gate will not be satisfied; the output coefficients of these gates will be equal to that of the inputs. While an AND gate can have different coefficients (c_1, c_2), the output coefficient will just be the product of its input coefficients($c_1 \cdot c_2$).

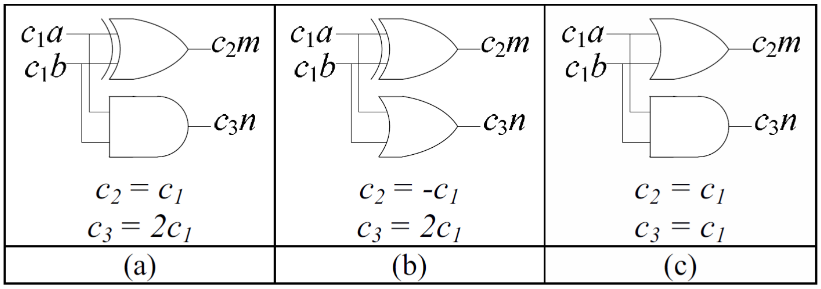

Figure 1: coefficient calculation based on gate structure

3.1 Remainder Generation

Given specification polynomial f and circuit implementation C as in figure 2, we need to rewrite f in terms of their gate polynomials derived from the circuit.The approach considers Jinpengs order derived from [3] wherein they start from the primary outputs, traverse the circuit to the primary inputs, and order the gates according to the their (reverse) topological levels denoted as the Reverse Topological Term Order($RTTO$).

For the given circuit, RTTO lexicographic order with variable order is given as:

$$\{z_3{>}z_2\}{>}\{z_1{>}S\}{>}\{z_0{>}M{>}N{>}O\}{>}\{a_0{>}a_1{>}b_0{>}b_1\}$$

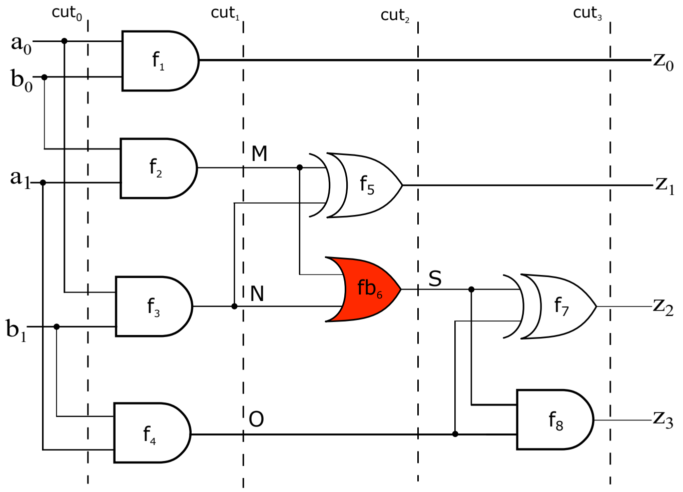

Figure 2: 2-bit integer multiplier

The specification rewriting for the circuit in figure 2 is as shown below:

spec f: $8.z_3 + 4.z_2 + 2.z_1 + z_0 - 4.a_1.b_1 - 2.a_1.b_0 - 2.a_0.b_1 - a_0.b_0$

$step1 : 8.(S.O) + 4.(S + O - 2.S.O) + 2.z_1 + z_0 - 4.a_1.b_1 - 2.a_1.b_0 - 2.a_0.b_1 - a_0.b_0$

$step1 : 4.S + 4.O + 2.z_1 + z_0 - 4.a_1.b_1 - 2.a_1.b_0 - 2.a_0.b_1 - a_0.b_0$

$step2 : 4.(M + N - M.N) + 4.O + 2.(M + N - 2.M.N) + z_0 - 4.a_1.b_1 - 2.a_1.b_0 - 2.a_0.b_1 - a_0.b_0$

$step2 : 4.M + 4.N + 4.O + 2.M + 2.N - 8.M.N + z_0 - 4.a_1.b_1 - 2.a_1.b_0 - 2.a_0.b_1 - a_0.b_0$

$step3 : 4.(a_1.b_0) + 4.(a_0.b_1) + 4.(a_1.b_1) + 2.(a_1.b_0) + 2.(a_0.b_1) - 8(a_1.b_0.a_0.b_1) + (a_0.b_0) - 4.a_1.b_1 - 2.a_1.b_0 - 2.a_0.b_1 - a_0.b_0$

$$remainder : R = 4.a_1.b_0 + 4.a_0.b_1 - 8.a_0.a_1.b_0.b_1 \tag{1}$$

3.2 Test Generation

Once we have a non-zero remainder, we need to identify all the test patterns which excite the bug in the design. The idea is to find assignments to all variables in the remainder such that it makes the decimal value of the remainder non-zero. The approach uses SMT solver by defining Boolean variables and considering signed/unsigned integer values as the total value of the remainder polynomial to identify such assignments. For example, for the remainder generated in equation(1), $(a_0 = 0, a_1 = 1, b_0 = 1)$ renders the equation to be a non-zero decimal value and hence is one of the test patterns.

3.3 Gate Pruning

Once we identify all the input assignments which excite the bug, the next step is to simulate these assignments against the specification polynomial to identify the points of difference in output bits. The idea is to prune the potential faulty gates list to a smaller set. For the given circuit in figure 2, the test patterns are simulated and compared against the specification polynomial to arrive at the faulty output bits set $\{z_2, z_3\}$. Once we have the affected output bits, the netlist is partitioned to find fanout free cones. The faulty gates in the construction of erroneous output z_2 are $\{f_2, f_3, f_4, fb_6, f_7\}$, while for z_3 are $\{f_2, f_3, f_4, fb_6, f_8\}$. For a single independent bug, the potential source of error gates are the intersection of these faulty gate lists which gives us the final pruned gate list:$P_g = \{f_2, f_3, f_4, fb_6\}$.

3.4 Bug Localization and Correction

Now, once we have pruned the gate list, the idea is to characterize the remainder as a pattern by rewriting the remainder in terms of the corresponding gate polynomials. Let's consider the circuit in figure 2 and assume that the AND gate f_1 is replaced with an XOR gate. Let us consider the effect of this bug from algebraic point of view: the equivalent algebraic value of the replaced XOR gate is $M = a_0 + b_0 - 2.a_0.b_0$, however the correct implementation with AND gate would have been $M^* = a_0.b_0$. Thus the difference between M and M^* will be $(a_0 + b_0 - 3.a_0.b_0)$ and will be observed in the remainder. Similarly for all the possible combinations of erroneous gate replacements, the pattern correlation in remainder will look as described in Table 1.

Once we have the remainder R from equation(1) and the potentially faulty gate list P_g, we need to match the remainder pattern against the suspicious patterns in Table 1. The steps to arrive at the remainder pattern matching is as show below:

1. We start from gate f_2 and compute $P_1 = -2.a_1 - 2.b_0 + 4.a_1.b_0$ and $P_2 = -2.a_1 - 2.b_0 + 6.a_1.b_0$. These patterns do not exist in R and hence gate 2 is correct and dictionary will be updated with $(M = 2.a_1.b_0)$.
2. Similarly, for gates f_3 and f_4, the dictionary will be updated with $(N = 2.a_0.b_1)$ and $(O = 4.a_1.b_1)$ respectively as their patterns do not match the remainder.
3. Now, for gate fb_6, the patterns can be written as $P_1 = 4.a_1.b_0 + 4.a_0.b_1 - 8.a_1.b_0.a_0.b_1$ and $P_2 = 4.a_1.b_0.a_0.b_1$
4. Since, the pattern P_1 matches the remainder pattern, the bug is an OR gate and based on the correction mentioned in Table 1, it should be replaced with an AND gate.

Table 1: Remainder pattern correction table

Suspicious Gate	**Appeared Remainder's Pattern**	**Solution**
AND(a,b)	$p_1 = -a - b + 2.a.b$	$s_1 : OR(a,b)$
	$p_2 = -a - b + 3.a.b$	$s_2 : XOR(a,b)$
OR(a,b)	$p_1 = a + b - 2.a.b$	$s_1 : AND(a,b)$
	$p_2 = a.b$	$s_2 : XOR(a,b)$
XOR(a,b)	$p_1 = a + b - 3.a.b$	$s_1 : AND(a,b)$
	$p_2 = -a.b$	$s_2 : OR(a,b)$

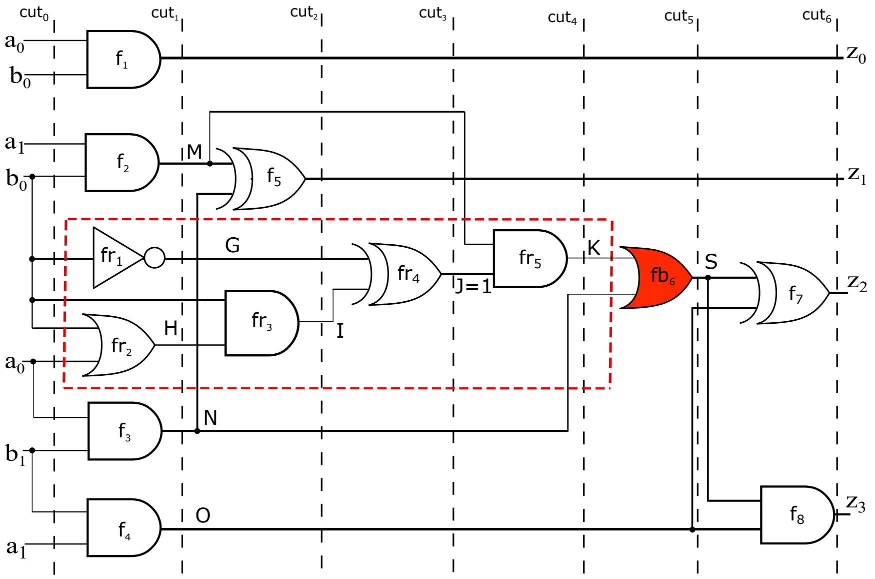

Figure 3: 2-bit integer multiplier with redundancy

4 Counterexample with Redundancy

Let us consider the circuit in figure 3, wherein we have added a redundant subcircuit as shown in the dotted red box which evaluates to '1'. The final output of redundancy K from fr_5 is still M, and hence retains the original functionality of the circuit in figure 2. Let us apply the same procedure on the circuit in figure 3 and try to identify the faulty gate.

4.0.1 Remainder Generation

Because the redundant circuit evaluates to '1', the remainder generated will still be the same and steps are as mentioned before.

$$R : 4.a_1.b_0 + 4.a_0.b_1 - 8.a_0.a_1.b_0.b_1$$

4.0.2 Test Generation

Since the remainder has the same exact terms and we don't have any new primary inputs introduced for the redundant circuit, the patterns to render a non-zero remainder will also remain the same.

4.0.3 Gate Pruning

As the faulty gate is not changed on the redundant circuit design, the affected output bits still remain the same (z_2, z_3). Since we have added redundant gates in the design cone, let us reevaluate the gate lists for affected output bits.

1. For z_2, the affected gate list is as follows:
   $\{f_2, fr_1, fr_2, f_3, f_4, fr_3, fr_4, fr_5, fb_6, f_7\}$,
   while for z_3:$\{f_2, fr_1, fr_2, f_3, f_4, fr_3, fr_4, fr_5, fb_6, f_8\}$.

2. The intersection of these lists gives us the potential fault gate list:
   $\{f_2, fr_1, fr_2, f_3, f_4, fr_3, fr_4, fr_5, fb_6\}$

4.0.4 Bug Localization and Correction

Let us reconstruct the gate polynomials from primary input side for the potential faulty gates to arrive at the matching pattern of the remainder. Let's also record the coefficient values in square brackets when we update the dictionary with the respective gate polynomials.

1. Gate f_2-AND patterns - $P_1 = -2.a_1 - 2.b_0 + 4.a_1.b_0$ and $P_2 = -2.a_1 - 2.b_0 + 6.a_1.b_0$. Since it doesn't match the remainder pattern, let's store $(M = a_1.b_0[\text{coeff - 2}]) \rightarrow (M = 2.a_1.b_0)$ in dictionary.

2. Since fr_1 patterns don't match, let's store $(G = 1 - b_0[\text{coeff - 1}]) \rightarrow (G = 1 - b_0)$ in dictionary.

3. similarly for fr_2, $(H = a_0 + b_0 - a_0.b_0[\text{coeff - 1}]) \rightarrow (H = a_0 + b_0 - a_0.b_0)$ is added to dictionary.

4. similarly for f_3, $(N = a_0.b_1[\text{coeff - 2}]) \rightarrow (N = 2.a_0.b_1)$ is added to dictionary.

5. similarly for f_4, $(O = a_1.b_1[\text{coeff - 4}]) \rightarrow (O = 4.a_1.b_1)$ is added to dictionary.

6. similarly for fr_3, $(I = H.b_0[\text{coeff - 1}]) \rightarrow (I = (a_0 + b_0 - a_0.b_0).b_0[\text{coeff - 1}]) \rightarrow (I = b_0)$ is added to dictionary.

7. similarly for fr_4, $(J = G + I - 2.G.I[\text{coeff - 1}]) \rightarrow (J = (1 - b_0) + b_0 - 2.(1 - b_0).b_0[\text{coeff - 1}]) \rightarrow (J = 1)$ is added to dictionary.

8. similarly for fr_5, $(K = M.J[\text{coeff - 2}]) \rightarrow (K = a_1.b_0[\text{coeff - 2}]) \rightarrow (K = 2.a_1.b_0)$ is added to dictionary.

9. similarly for last gate fb_6, the two patterns are $P_1 = K + N - 2.K.N = 2.a_1.b_0 + 2.a_0.b_1 - 8.a_0.a_1.b_0.b_1$ and $P_2 = K.N = 4.a_0.a_1.b_0.b_1$.

   Despite having a close resemblance to remainder R, P_1 still doesn't have the correct coefficients to match the remainder. Hence, $(S = K + N - K.N[\text{coeff - 4}]) \rightarrow (S = a_1.b_0 + a_0.b_1 - a_0.a_1.b_0.b_1[\text{coeff - 4}]) \rightarrow (S = 4.a_1.b_0 + 4.a_0.b_1 - 4a_0.a_1.b_0.b_1)$ is added to dictionary and we have run out of faulty gate list to analyze the remainder pattern matching.

This shows that, when there is redundancy in the circuit, which doesn't alter the function implemented by the circuit, the approach fails to recognize the faulty gate sue to the ambiguity in coefficient computation, which results in remainder pattern not being matched.

References

[1] F. Farahmandi and P. Mishra. Automated Test Generation for Debugging Arithmetic Circuits. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016.

[2] S. Ghandali, C. Yu, D. Liu, W. Brown, and M. Ciesielski. Logic Debugging of Arithmetic Circuits. In *IEEE Computer Society Annual Symposium on VLSI*, 2015.

[3] J. Lv, P. Kalla, and F. Enescu. Efficient Gröbner Basis Reductions for Formal Verification of Galois Field Multipliers. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012.