# Debugging Methods Through Identification of Appropriate Functions for Internal Gates

Kosuke Oshima, Takeshi Matsumoto, Masahiro Fujita

# Debugging Methods through Identification of Appropriate Functions for Internal Gates

Kosuke Oshima[1], Takeshi Matsumoto[2], and Masahiro Fujita[3]

[1] Dept. of Electrical Engineering and Information Systems, The University of Tokyo
oshima@cad.t.u-tokyo.ac.jp
[2] Dept. of Electronics and Information Engineering, Ishikawa National College of Technology
matsumoto@ishikawa-nct.ac.jp
[3] VLSI Design and Education Center, The University of Tokyo
fujita@ee.t.u-tokyo.ac.jp

**Abstract.** In this chapter, we propose methods for correcting gate-level designs by identifying appropriate logic functions for internal gates. We introduce programmable circuits, such as look up table (LUT) and multiplexer (MUX) to the circuits under debugging, in order to formulate the correction processes mathematically. There are two steps in the proposed methods. The first one is to identify sets of gates and their appropriate inputs whose functions are to be modified. The second one is to actually identify logic functions for the correction by solving QBF (Quantified Boolean Formula) problems with repeated application of SAT solvers. There are a number of bugs which cannot be corrected unless the inputs of the gates to be modified are changed from the original ones, and the selection of such additional inputs is a key for effective debugging. We show a couple of methods by which appropriate inputs to the gates can be efficiently identified. Experimental results for each such a method as well as their combinations targeting benchmark circuits as well as industrial ones are shown.

**Keywords:** Gate-level circuit, Design debugging, Programmable circuit

## 1 Introduction

Thanks to the advancement of semiconductor technology, VLSI designs keep becoming larger and more complicated continuously. Now verifying such huge VLSI chips is a big challenge and needs lots of human efforts and time, which can approach to 80 % or more of the total design time. Moreover, some bugs may not be detected in the verification processes before fabrication and are only recognized by running an actual chip after fabrication. In such cases, re-spin of the whole design processes, including very time-consuming physical and timing design processes, must be performed again. Re-spins often happen due to the insufficient time for verification and debugging in pre-silicon design phases. Due to the design schedule, sometimes the verification and debugging efforts must be terminated before the design is fully examined. If the verification and debugging processes become more efficient and effective, significantly more

bugs could be detected and corrected before fabrication. Usually, more than a half of the verification time is spent for correcting the buggy portions of the designs rather than identifying them, since debugging is much less automated than checking correctness of the designs. Thus, automating and shortening the debugging processes is now one of the most important issues in VLSI designs. The efficiency of verification is very important to find more bugs or most bugs in a shorter time, and debugging is equally or more important since same or longer time is now spent for correcting the buggy portions of the design after recognizing that the design is not correct.

In this chapter, we focus on debugging gate-level designs. We assume existence of a specification in terms of golden models in RTL or in gate level. Our method tries to let a given circuit under debugging behave equivalently to the specification through modifications inside the circuit. To this end, our method tries to identify the appropriate new functions for some of the internal gates in the circuit, so that the circuit as a whole can have an equivalent logic function to the specification. For the purpose of formulating the debugging process mathematically, we introduce some amount of programmability in the circuit under debugging and find a way to appropriately program it for correction. Note that, after identifying such appropriate functions for internal gates, those gates are assumed to be completely replaced with new gates corresponding to the those functions. In other words, programmability is introduced only for mathematical modeling and is nothing to do with actual implementations.

In [1], Yamashita et al. proposed Partially-Programmable Circuit (PPC) where programmable circuits such as look-up tables (LUTs) and multiplexers (MUXs) are added to the original circuit in order to correct bugs and/or defects in fabricated chips. While their purpose of introducing programmability is to make chips re-programmable or correctable in post-silicon (*patching* in their words), our purpose in this work is to find correct logic functions for internal gates by which the entire designs can be rectified in pre-silicon design phases. Therefore, in principle, we can freely add programmable circuits in the target circuit under debugging without considering physical implementation. Once a circuit is corrected, its logical and physical design can be performed again in such a way that the programmable circuits that are introduced for debugging are replaced with appropriate standard logic gates. If it is better to keep physical structures of the circuit similar as much as possible in order to avoid performing complicated physical design processes again, the corrected logic may be implemented utilizing various Engineering Change Order (ECO) techniques.

Debugging consists of two processes: locating the suspicious portions in the designs and correcting them through replacements with appropriate sets of gates. In the locating process, designers try to find locations of bugs (or candidate locations) which should be the root cause of the bugs. Then, they modify logic functions at those possibly buggy locations in the correcting process. There exist researches for locating bugs in gate-level circuits such as path tracing and SAT-based diagnosis[8]. Note that the methods such as [8] examine the designs only with given counterexamples and do not refer to specifications. This makes the analysis much simplified, but appropriate locations for all the bugs in the designs may not be found, as counterexamples may be related only to the specific bugs in the designs. Moreover, it is very difficult to identify a small set of logic gates that need to be modified for correction, especially when there are multiple

bugs in a design. In addition, some methods such as [8] require designers to come up with a new logic function that can correct the current buggy one, which may take a long time if that is a manual process. Since the methods such as [8] assume that any primary input can be used when creating the functions for correction, in the worst case, the circuit size can become almost doubled. Based on the above discussion, there is a large demand for methods which generate logic functions for the set of logic gates identified in the bug locating processes, by which the entire designs become correct.

The basic idea of our proposed debugging methods is to correct a circuit under debugging by finding another logic function for a set of gates that is identified as a bug location, in such a way that the entire circuit becomes equivalent to its specification[9]. In the methods, the logic function for each gate for correction has the same set of inputs as the original gate. In other words, our method tries to replace each of the original (possibly) buggy gates with a different logic gate having the same input variables. So, the new gate to be used for replacement may have to realize a complicated logic function with the same inputs and can only be implemented with a set of simple gates, such as, NAND, NOR, etc. As described in the following sections, we utilize an existing method proposed in [2, 7] to efficiently generate logic functions for programmable circuits. There are, however, bugs which cannot be corrected if the input variables of the gates remain the same. In such cases, we need to add an additional input variable to LUTs or MUXs. When the number of variables in a circuit is very large, it is not practical to check all the variables one by one. To quickly find variables which cannot improve the chance of getting a correct logic when they are connected to LUTs or MUXs, we introduce a necessary condition that should be satisfied by each variable in order to improve the chance of correction. We also propose an efficient selection method based on that condition.

The contributions of this chapter are summarized as follows.

- We propose a method to find a correction by introducing programmability in a circuit under debugging based on LUTs or MUXs and mathematically formulating the debugging problem.
- To deal with cases where a correction is impossible due to a lack of certain input variables to LUTs or MUXs, we propose a method to filter out variables which cannot contribute to improving the possibility of correction.
- Through the experiments with various designs and bugs including industrial ones, we show that our proposed method can provide a correction in a short time in many cases.

The remainder of this chapter is organized as follows. In Section 2, we introduce an existing method to efficiently find a logic function of each programmable circuit for correction. In Section 3, our proposed method is described in detail. In Section 4, we show experimental results with bugs in an industrial on-chip network circuit and ARM Cortex microprocessor designs as well as benchmark circuits. Section 5 gives ideas for extensions of the proposed debugging methods with preliminary experimental results. Finally, in Section 6, concluding remarks and future directions are discussed.

## 2 Related Work: Finding a Configuration of LUTs using Boolean SAT Solvers

For easiness of explanation, in this chapter, we assume the number of output for the target buggy circuit is one. That is, one logic function in terms of primary inputs can represent the logic function for the entire circuit. This makes the notations much simpler, and also extension for multiple outputs is straightforward. Also, variables in this chapter are mostly vectors of individual ones.

As there is only one output in the design, a specification can be written as one logic function with the set of primary inputs as inputs to the function. For a given specification $SPEC(x)$ and an implementation with programmable circuits $IMPL(x, v)$, where $x$ denotes the set of primary input variables and $v$ denotes the set of variables to configure programmable circuits inside, the problem is to find a set of appropriate values for $v$ satisfying that $SPEC$ and $IMPL$ are logically equivalent. This problem can be described as QBF (Quantified Boolean Formula) problem as follows:
$\exists v.\forall x.SPEC(x) = IMPL(x, v)$.
That is, with appropriate values for $v$, regardless of input values (values of $x$), the circuits must be equivalent to the specification (i.e., the output values are the same), which can be formulated as the equivalence of the two logic functions for the specification and the implementation. There are two nested quantifiers in the formula above, that is, existential quantifiers are followed by universal quantifiers, which are called QBF in general. Normal SAT formulae have only existential quantifiers and no universal ones.

In [2], CEGAR(Counter-Example Guided Abstraction Refinement) based QBF solving method is applied to the circuit rectification problem. Here, we explain the method using 2-input LUT for simplicity, although LUT having any numbers of inputs can be processed in a similar way.

Logic functions of a 2-input LUT can be represented by introducing four variables, $v_{00}, v_{01}, v_{10}, v_{11}$, each of which corresponds to the value of one row of the truth table. Those four variables are multiplexed with the two inputs of the original gate as control variables, as shown in Figure 1. In the figure a two-input AND gate is replaced with a two-input LUT. The inputs, $t_1$, $t_2$, of the AND gate becomes the control inputs to the multiplexer. With these control inputs, the output is selected from the four values, $v_{00}, v_{01}, v_{10}, v_{11}$. If we introduce $M$ of 2-input LUTs, the circuit has $4 \times M$ more variables than the variables that exist in the original circuit. We represent those variables as $v_{ij}$ or simply $v$ which represents a vector of $v_{ij}$. $v$ variables are treated as pseudo primary inputs as they are programmed (assigned appropriate values) before utilizing the circuit. $t$ variables in the figure correspond to intermediate variables in the circuit. They appear in the CNF of the circuits for SAT/QBF solvers.

If the logic function at the output of the circuit is represented as $f_I(v, x)$ where $x$ is an input variable vector and $v$ is a program variable vector, after replacements with LUTs, the QBF formula to be solved becomes:
$\exists v.\forall x.f_I(v, x) = f_S(x)$,
where $f_S$ is the logic function that represents the specification to be implemented. Under appropriate programming of LUTs (assigning appropriate values to $v$), the circuit behaves exactly the same as specification for all input value combinations.

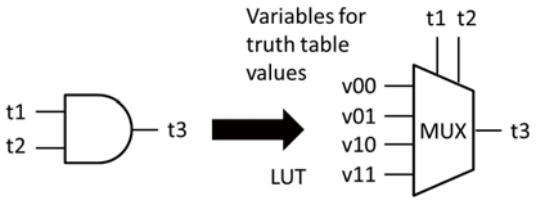


**Fig. 1.** LUT is represented with multiplexed four variables as truth table values.

Although this can simply be solved by any QBF solvers theoretically, only small circuits or small numbers of LUTs can be successfully processed [2]. Instead of doing that way, we here like to solve given QBF problems by repeatedly applying normal SAT solvers using the ideas shown in [3, 4].

Basically, we solve the QBF problem only with normal SAT solvers in the following way. Instead of checking all value combinations on the universally quantified variables, we just pick up some small numbers of value combinations and assign them to the universally quantified variables. This would generate SAT formulae which are just necessary conditions for the original QBF formulae. Note that here we are dealing with only two-level QBF, and so if universally quantified variables get assigned actual values (0 or 1), the resulting formulae simply become SAT formulae. The overall flow of the proposed method is shown in Figure 2. For example, if we assign two combinations of values for $x$ variables, say $a1$ and $a2$, the resulting SAT formula to be solved becomes like: $\exists v.(f_I(v, a1) = f_S(a1)) \wedge (f_I(v, a2) = f_S(a2))$. Then we can just apply any SAT solvers to them. If there is no solution, we can conclude that the original QBF formulae do not have solution neither. If there is a solution found, we need to make sure that it is a real solution for the original QBF formula. Because we have a solution candidate $v_{assigns}$ (these are the solution found by SAT solvers) for $v$, we simply make sure the following:
$\forall x.f_I(v_{assigns}, x) = f_S(x)$.
This can be solved by either usual SAT solvers or combinational equivalence checkers. In the latter case, circuits with tens of millions of gates may be processed, as there have been conducted significant amount of researches for combinational equivalence checkers which utilize not only state-of-the-art SAT techniques but also various analysis methods on circuit topology. If they are actually equivalent, then the current solution is a real solution of the original QBF formula. But if they are not equivalent, a counterexample, say $x_{sol}$, is generated and is added to the conditions for the next iteration:
$\exists v.(f_I(v, a1) = f_S(a1)) \wedge (f_I(v, a2) = f_S(a2)) \wedge (f_I(v, x_{sol}) = f_S(x_{sol}))$.
This solving process is repeated until we have a real solution or we prove the non-existence of solution. In the left side of Figure 2, as an example, the conjunction of the two cases where inputs/output values are $(0, 1, 0)/1$ and $(1, 1, 0)/0$ is checked if satisfiable. If satisfiable, this gives possible solutions for LUTs. Then using those solutions for LUTs, the circuit is programmed and is checked to be equivalent with the

specification. As we are using SAT solvers, usually non-equivalence can be made sure by checking if the formula for non-equivalence is unsatisfiable.

Satisfiability problem for QBF in general belongs to P-Space complete. In general QBF satisfiability can be solved by repeatedly applying SAT solvers, which was first discussed under FPGA synthesis in [5] and in program synthesis in [6]. The techniques shown in [3, 4] give a general framework on how to deal with QBF only with SAT solvers. These ideas have also been applied to so called partial logic synthesis in [7].

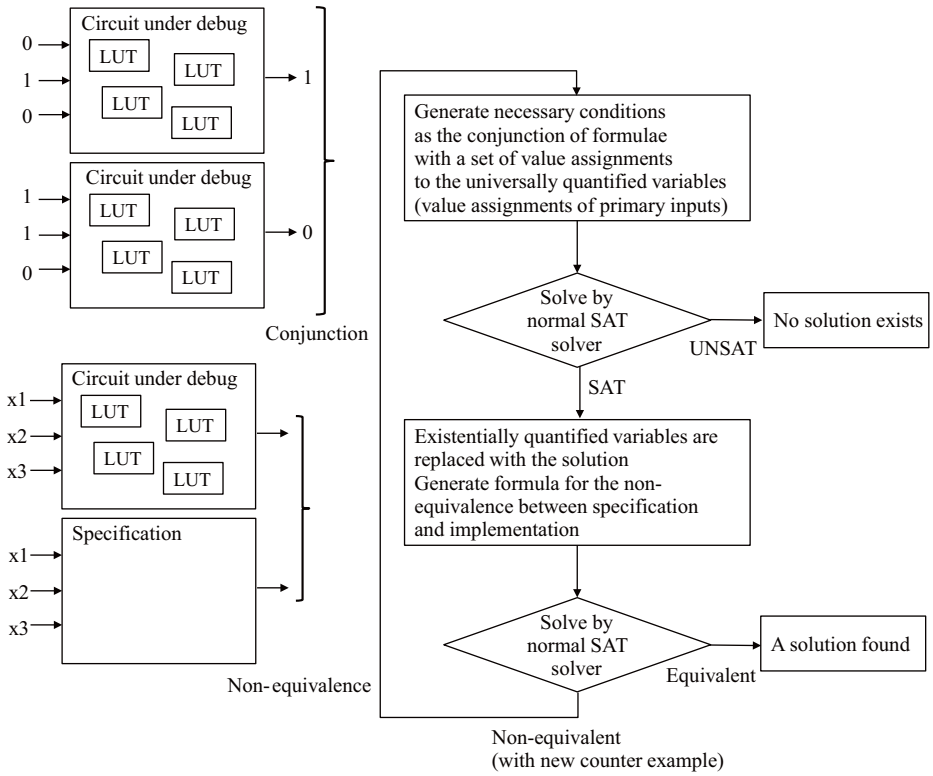


**Fig. 2.** Overall flow of the rectification method in [2]

## 3 Our Proposed Method to Correct Gate-Level Circuits

### 3.1 Overall Flow

Figure 3 shows an overall flow of our proposed correction method. Given

– a specification,

– an implementation circuit that has bugs, and
– a set of candidate locations of the bugs,

the method starts with replacing each logic gate corresponding to a candidate bug location with an LUT. Each inserted LUT has the same set of input variables as its original gate. Then, by applying the method in [2, 7], we try to find a configuration of the set of LUTs so that the specification and the implementation become logically equivalent. Once such a configuration is found, it immediately means we get a logic function for correction. Then, another implementation will be created based on the corrected logic function, which may require re-synthesis or synthesis for ECO. Although the method to compute a configuration of LUTs for correction in [2, 7] is relatively more efficient than most of the other methods, it can solve up to hundreds of LUTs within a practical runtime. Therefore, it is not practical to replace all of the gates in the given circuit with LUTs, and the number of LUTs inserted into the implementation influence a lot on the runtime. In order to obtain candidate locations of bugs, existing methods such as [8] can be utilized. In this work, we employ a simple heuristic, which is something similar to the path tracing method that all gates in logic cones of erroneous primary outputs are replaced with LUTs when they are within a depth of $N$ levels from the primary outputs. Figure 4 shows an example of such introduction of LUTs. In this figure $N = 2$. In the experiments described in Section 4, $N$ is set to 5. This number is determined through experiments. If the number is larger, there are more chances for the success of corrections. On the other hand, if the number is smaller, we can expect faster processing time.

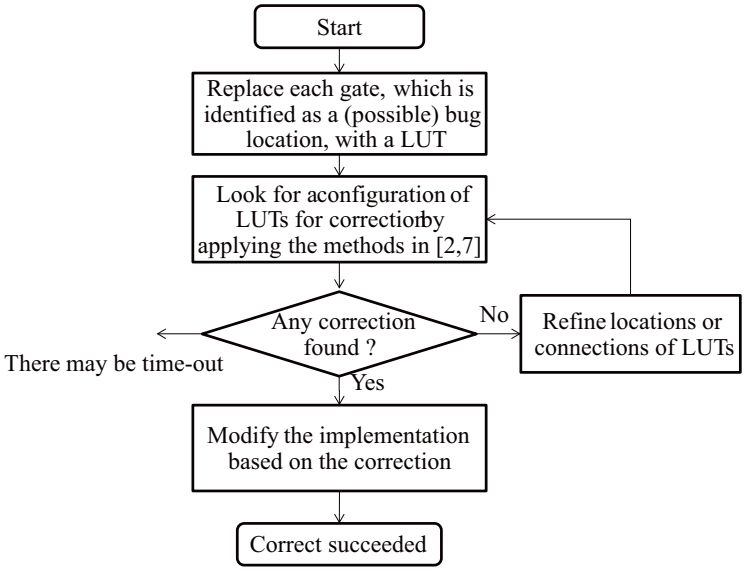


**Fig. 3.** An overall correction flow

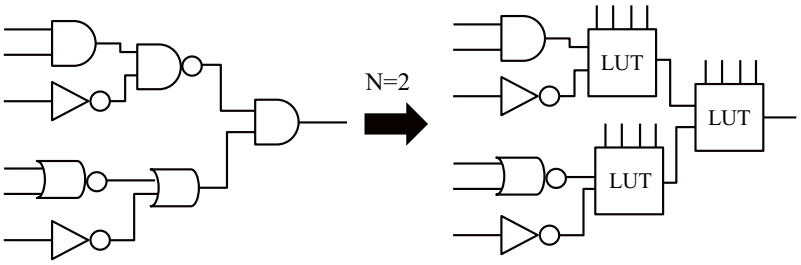


**Fig. 4.** An example of LUT insertions

There can be cases where any correction cannot be found for a given implementation with LUTs. There can be varieties of reasons on the failure. It may be due to the wrong selection of the target gates to be replaced, the inputs to LUTs are not sufficient, or other reasons. In this chapter, we assume that bugs (or portions that are implemented differently from designers' intention or specification) really exist within the given candidate locations. That is, we need to add more variables to the inputs of the LUTs to increase the chances of corrections, which is discussed in the next subsection.

### 3.2 Adding Variables to LUT Inputs

As mentioned above, there are bugs that cannot be corrected with LUTs having the same set of input variables as their original gates, if so called "missing wire" bugs in [12] are happening. Figure 5 shows a simple example. In this example, the logic function of an implementation generates $A \wedge B \wedge C$, while its specification is $A \vee B \vee C \vee D$. With an LUT whose inputs are $A, B, C$ that replaces the original AND gate in the incorrect implementation, we cannot get any configuration of its truth table for correction, since $D$ is essential to the correct logic function. In general, assuming that bugs really exist within the gates that are replaced with LUTs, the reason why we cannot obtain any correction is due to the lack of variables that should be connected to appropriate LUTs. Therefore, what we need to do in the refinement phase in Figure 3 is to add extra variables to LUT inputs and try to find a correction again. If we inappropriately add a set of variables to inputs of some LUTs, however, it simply results in no solution in the next iteration of the loop in Figure 3. The numbers of ways to add extra variables to input of LUTs are large, and we cannot check one by one. In our method, we try to correct the implementation by adding as small numbers of variables as possible. First, all possible ways to add one variable to LUTs are tried. If no correction can be found, then the method looks for correction with two additional variables to one or two LUTs. Basically, we continue this process until we find corrections.

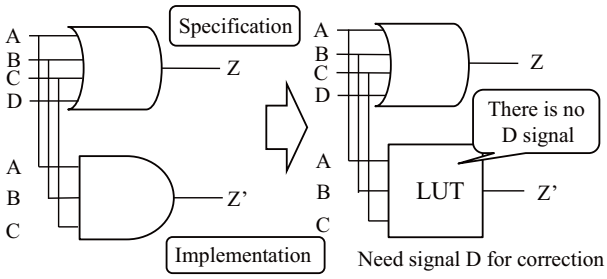


**Fig. 5.** An example bug that cannot be corrected with LUTs having the same inputs

### 3.3 Using MUXs to Examine Multiple Additional Variables

As discussed above, the method looks for any correction by adding variables to the inputs of LUTs. Even if only one variable is added to the inputs of some LUT, we need to iterate the loop in Figure 3 many times until a correction is found or a proof of no solution is obtained. For a large circuit, the number of iterations may be too large even for the case of adding one variable to an LUT. To make this process more efficient, we introduce a multiplexer and connect multiple variables, which are candidates to be added to an LUT, to its inputs. The output of the MUX and the additional input of the LUT are connected as shown in Figure 6. Then, we can select a variable to be added to the LUT by appropriately assigning values to the control variables of the MUX.

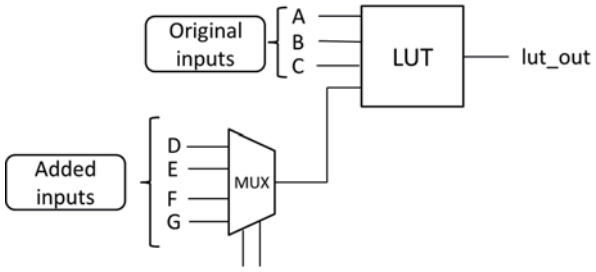


**Fig. 6.** Additional input variables to an LUT

Figure 6 shows how multiplexers work for examining candidate variables that may need to be added to the inputs of the LUT in order to get a correction. The LUT in the example originally has three inputs, $A$, $B$, and $C$, which means this LUT is supposed to be replaced with some 3-input logic gate for corrections. Suppose that we want to examine multiple variables to be added as an input of the LUT at the same time, instead of examining one by one. Using the MUX in the example, we can examine four additional candidate variables $D$, $E$, $F$, and $G$ at one iteration. Here, we need to treat the control

variables as program variables, same as the ones in the LUTs. If any correction is found, the corresponding values of the control variables identify a variable for addition. That is, if it becomes an input of the LUT connected to the MUX, the implementation can be equivalent to its specification. Otherwise, all variables connected to inputs of the LUT cannot make the incorrect implementation equivalent to its specification. A straightforward way to realize something similar is to introduce LUTs having larger numbers of inputs rather than using MUX. This is definitely more powerful in terms of the numbers of function which can be realized at the output of the LUTs. In the example shown in Figure 6, instead of using a MUX, an LUT having seven inputs may be used, and that LUT can provide many more distinct functions for possible corrections. The problem, however, is the number of required program variables. If we use a MUX in the example, we need $2^4 + 2 = 18$ variables. If we use a seven-input LUT, however, we need $2^7 = 128$ variables, which may not be practical with our methods if we deal with multiple of such cases simultaneously.

Even when MUXs are used to examine multiple variables at the same time, we should be aware of the increase of the number of program variables. As can be seen in [2, 7], larger numbers of program variables increase runtime for finding a correction, which corresponds to the runtime spent for each iteration of the loop in Figure 3. Note that one iteration in Figure 3 may include many iterations in Figure 2. In the experiments, we show a case study with varieties of numbers of inputs to MUX.

### 3.4 Filtering Out Variables Based on Necessary Condition

When a variable is added to an input of an LUT, it may or may not be an appropriate variable to correct the target bug. Even with the more efficient method using MUXs described above, we should not try to examine a variable which cannot correct the bug. In this subsection, we propose a method for filtering out such non-useful variables from the set of candidate variables that can be connected to inputs of LUTs by utilizing necessary conditions on the correctability.

**Necessary condition for the variables to be added.** For simplicity, the following discussion assumes that there is one LUT added to an implementation circuit. It can be easily extended to the cases of a set of multiple LUTs where each LUT does not have any other LUTs in its fan-in cone (i.e. an LUT depends on other LUTs). However, we here omit such cases.

When no correction is found, which corresponds to taking "NO" branch in Figure 3, we cannot correct an implementation under debugging with the current LUT, which is the only LUT added, with its current input variables. The reason why there is no correction (i.e. no configuration of LUTs works correctly) is that the LUT outputs the same value for different two input values to the LUT. This happens when "No solution" is reached in Figure 2. Figure 7 explains the situation. In the figure, $x_i$ is an input pattern added in one of the previous iterations of the process shown in Figure 2, and $x_j$ is the pattern that is added as a result of the last iteration. Then, there can be situations where the following two conditions are satisfied.

1. For a pair of primary input patterns $x_i$ and $x_j$, the input values to the LUT $l_{in}(x_i)$ and $l_{in}(x_j)$ are the same, where $l_{in}$ represents a logic function that determines an input value to the LUT for a given primary input pattern. Therefore, the output values from the LUT are also the same, that is, $l_{out}(x_i) = l_{out}(x_j)$.
2. In order to make the implementation equivalent to the specification for both $x_i$ and $x_j$, that is, $f_I(x_i) = f_S(x_i) \wedge f_I(x_j) = f_S(x_j)$, $l_{out}(x_i)$ and $l_{out}(x_j)$ must be different, where $f_S$ and $f_I$ denote logic functions of the primary outputs of the specification and the implementation, respectively.

Note that $f_S(x_i)$ can be a different value from that of $f_S(x_j)$. With the conditions, there is no way to have an LUT configuration that satisfies the specification for both $x_i$ and $x_j$ at the same time. In this case, it cannot make an LUT configuration for both $x_i$ and $x_j$ even if we add a variable $v$ to the LUT that has the same value for $x_i$ and $x_j$ ($v(x_i) = v(x_j)$), since $l_{out}(x_i)$ and $l_{out}(x_j)$ are still the same. This is because the output of the LUT can be represented as $l_{out}(l_{in}(x), v(x))$ for a primary input pattern $x$ and $l_{in}(x_i) = l_{in}(x_j) \wedge v(x_i) = v(x_j)$ implies $l_{out}$ are equivalent for $x_i$ and $x_j$ for any configuration of the LUT.

The observation above suggests that we must not add a variable to the LUT inputs if it has the same value for $x_i$ and $x_j$. It gives us a necessary condition that the added variable to an LUT must have different values for $x_i$ and $x_j$. If this necessary condition is satisfied, there is an LUT configuration where $l_{out}(x_i) \neq l_{out}(x_j)$ is satisfied, which is a requirement to make $f_S = f_I$ for both $x_i$ and $x_j$. Note that $f_S = f_I$ may not be satisfied even $l_{out}$ is different for the two input patterns.

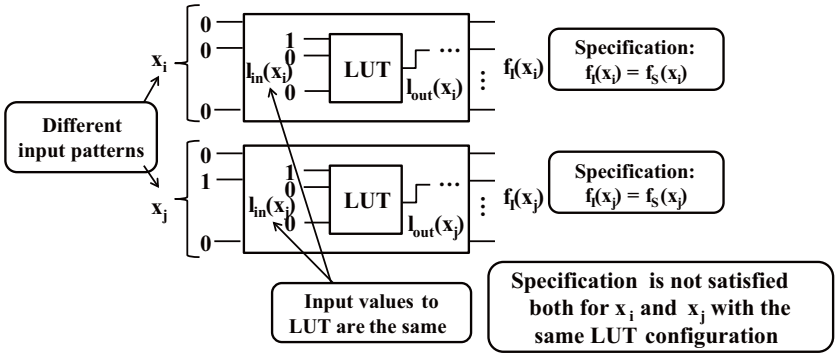


**Fig. 7.** Reason of no correction

Figure 8 shows how to make the output of the LUT different by adding a variable that satisfies the necessary condition. Here, we denote the added variable to the LUT as $A(x)$, where $x$ is the primary input variables. An LUT configuration with its input $l_{in}(x)$ and $A(x)$ is represented by $l'_{out}(x)$, which is rewritten as $l'_{out}(x) = P_{\bar{A}}(l_{in}) * \bar{A} + P_A(l_{in}) * A$, where $P_{\bar{A}}(l_{in})$ and $P_A(l_{in})$ represents truth table values for $l_{in}$ when $A = 0$ and $A = 1$, respectively. This is nothing but Shannon's expansion of $l'_{out}$. If

the added variable $A(x)$ that satisfies the necessary condition takes 1 for $x_i$ and 0 for $x_j$, we can make an LUT configuration satisfying $l'_{out}(x_i) \neq l'_{out}(x_j)$ by setting the two truth tables $P_{\bar{A}}$ and $P_A$ appropriately. For the case of $x_i = 0$ and $x_j = 1$, an LUT configuration can be obtained in a similar way.

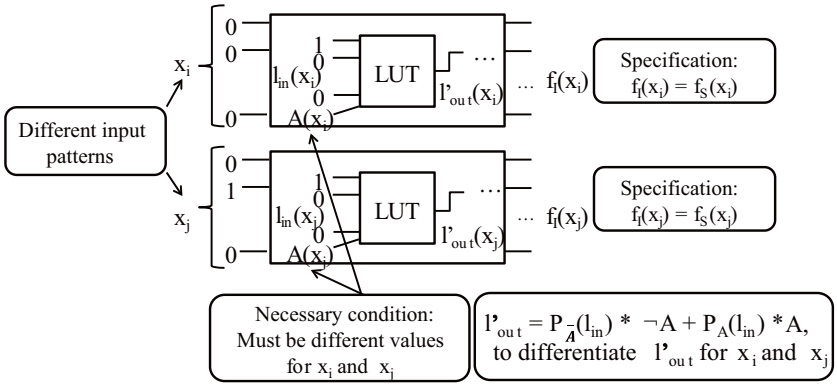


**Fig. 8.** Adding a variable satisfying a necessary condition

Based on the discussion above, we can filter out variables from candidates when they have the same value for both $x_i$ and $x_j$. Now we show an example of such filtering. Figure 9 (a) is the specification which is $Z = A \vee B \vee C \vee D$. Here we assume that this is one of the specifications and there are other outputs in the target circuit. Now assume that a wrong implementation is generated as shown in Figure 9 (b). Here the output only depends only on $C$ and $D$, which is clearly wrong. For the input values where all of inputs are 0, this implementation looks correct as it generates the same output value, 0, as the specification. Note that the implementation has more gates in the circuit in order to realize the other outputs which are not shown in the figure.

Then we find a counterexample, which is $A = 0, B = 1, C = 0, D = 0$ as shown in Figure 10 (a). For these values, the correct output value is 1, but the value of the output in the implementation is 0 as seen from Figure 10 (b). Our debugging method first replaces the suspicious gate, the OR gate, with an LUT as shown in Figure 10 (c). Unfortunately, there is no configuration for the LUT which makes the implementation correct, and so we need to add an variable to the LUT. Now we have two candidates, $t_1$ and $t_2$ as shown in Figure 10 (d). The necessary condition discussed in the above requires that the value of the variable must be different between the two cases, $A = 0, B = 0, C = 0, D = 0$ and $A = 0, B = 1, C = 0, D = 0$. From this condition, the variable $t_1$ is eliminated and the variable $t_2$ is selected.

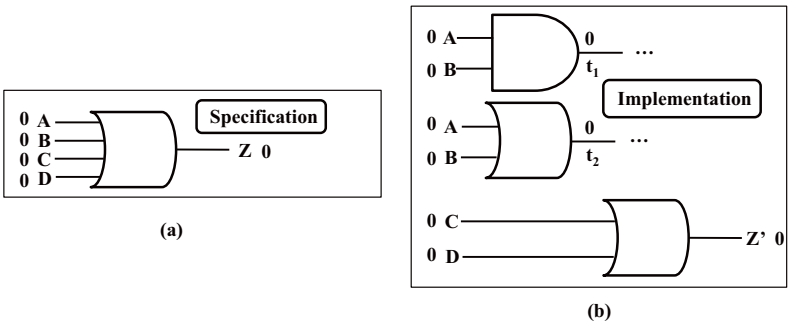


**Fig. 9.** An example specification and its buggy implementation

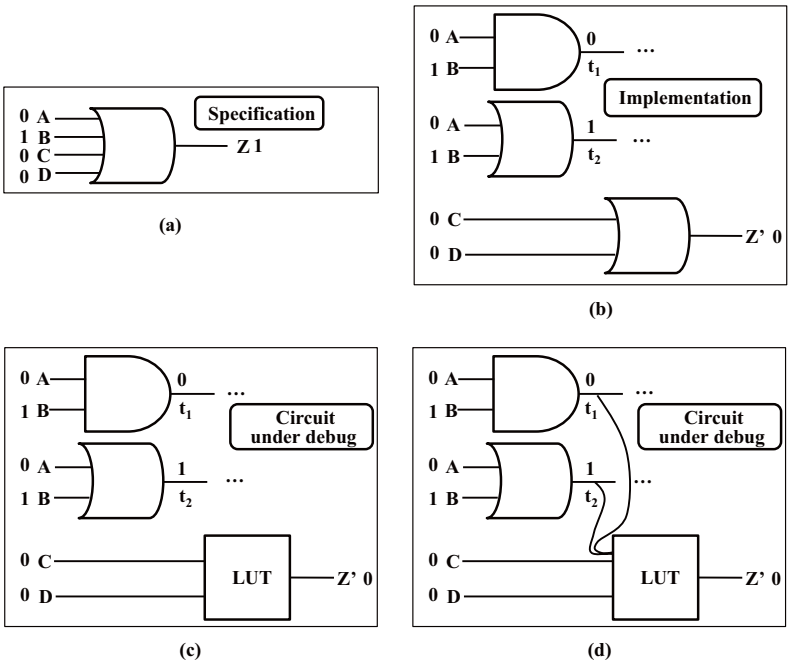


**Fig. 10.** A debugging process for the design in Figure 9 with a necessary condition

**An improved flow with filtering variables.** Figure 11 shows an improved flow with variable filtering based on the necessary condition discussed above. When no correction is found for all the input patterns so far, the method searches for a set of variables that can be added to inputs of an LUT. During this search, variables which do not satisfy the necessary condition are filtered out. This consists of the following two steps:

1. Find an input pattern $x_i$ that is added in one of the previous iterations and has the same input values of an LUT as those of the lastly added pattern $x_j$.
2. Find a variable having different values for $x_i$ and $x_j$.

As a result, the method tries to add a variable satisfying the necessary condition to inputs of some LUT. Then, with the added LUT input, the method looks for another correction $v$ by applying CEGAR based method in [2, 7].

If this filtering method is applied with the method using MUXs to examine multiple variables simultaneously that is described in Section 3.3, it needs to pick up $N$ variables, where $N$ is the total number of input variables to MUXs.

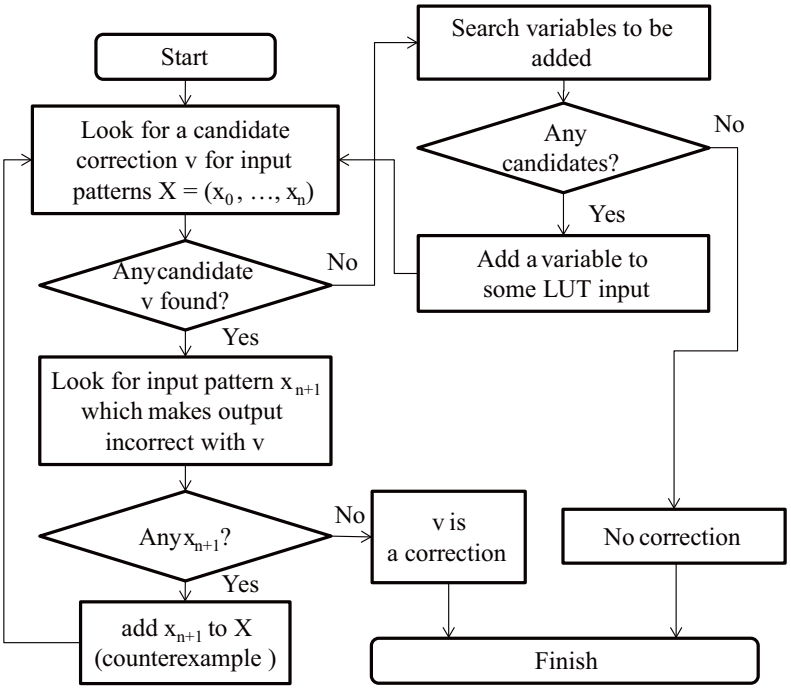


**Fig. 11.** An bypass flow including filtering out variables

# 4 Experimental Results

## 4.1 Experimental Setup

Four sets of experiments are conducted in order to evaluate our debugging methods proposed in this chapter. We use the following circuits for the experiments: ISCAS85 benchmark circuits, an industrial on-chip network circuit ("Industrial"), and an ARM Cortex microprocessor ("ARM processor"). While ISCAS85 circuits are combinational ones, the last two circuits are sequential ones. All are in gate level designs except for the last experiment which deals with bugs in RTL designs. Table 1 shows the characteristics of these circuits. In order to apply our method, sequential circuits needs to be time-frame expanded. The number of expanded time-frames (i.e. clock cycles for examinations) are shown in the second column for Industrial and ARM processor.

We use PicoSAT[10] as a SAT solver. In order to convert the netlists written in Verilog into SAT formulae, we use ABC[11] and AIGER[13]. All experiments reported in this section are run on a machine with Intel Core 2 Duo 3.33GHz CPU and 4GB Memory.

**Table 1.** Characteristics of circuits

| | # of expansion | Inputs | Outputs | Gates |
|---|---|---|---|---|
| ISCAS85 benchmarks | | | | |
| c499 | | 202 | 41 | 32 |
| c880 | | 383 | 60 | 26 |
| c1355 | | 546 | 41 | 32 |
| c1908 | | 880 | 33 | 25 |
| c2670 | | 1193 | 233 | 140 |
| c3540 | | 1669 | 50 | 22 |
| c5315 | | 2307 | 178 | 123 |
| c7552 | | 3512 | 207 | 108 |
| Others | | | | |
| Industrial | 3 | 1201 | 1216 | 8289 |
| ARM processor | 1 | 895 | 923 | 4666 |

## 4.2 Simultaneous examination on multiple variables using multiplexers

First, we perform an experiment with our method that introduces multiplexers (MUXs) into a circuit under debugging so that multiple extra variables are connected to LUTs through MUXs. In this experiment, we identify the erroneous primary outputs through simulation, and replace all gates in their logic cones within the depth of 5 levels from the erroneous primary outputs with LUTs. Then, we insert a $N$-input MUX to the circuit, and its output is connected to all LUTs. We randomly choose sets of variables out of

all primary inputs of the circuit to be debugged, and they are connected to the inputs of MUXs.

If no solution for correction can be found, we replace all the input variables to MUXs with another set of variables that are not examined yet, and execute the method again. In this experiment, the runtime is limited up to 5 hours.

The results are shown in Table 2. $N$, the number of inputs to MUX, varies from 1 to 256. $N = 1$ means no MUX, in other words, a variable is directly added to inputs of all LUTs. "Change inputs" represents the number of variable sets that are examined for correction. If this number is $M$, $N \times M$ variables are examined in total. As can be seen in the table, we need to run the method in $[2, 7]$ only a few times when the number of MUX inputs is 64 or 256. "Time" shows the total runtime. We can see the runtime for 256-input MUX is the shortest in both circuits. Also, it is notable that we cannot find a correction within 5 hours without MUX, since a lot of iterations are required in order to check many variables one by one.

**Table 2.** Experimental results of simultaneous examination of candidate variables using MUXs

| | Inputs of MUX | Change inputs | Time (sec) |
|---|---|---|---|
| Industrial | 1(no MUXs) | - | Timeout (-) |
| | 16 | 15 | 5281 |
| | 64 | 4 | 12794 |
| | 256 | 1 | 211 |
| ARM processor | 1(no MUXs) | - | Timeout (-) |
| | 16 | 8 | 11204 |
| | 64 | 2 | 8857 |
| | 256 | 1 | 5909 |

### 4.3 Candidate variable filtering using the necessary condition

Next, we conduct another experiment to evaluate our method in terms of how well the necessary conditions works. In this experiment, only an incorrect gate is replaced with an LUT. The candidates of variables are all variables in the circuit under debugging. For this experiment, we need to record the values of internal variables for all input patterns. For this purpose, we use Icarus Verilog simulator [14].

The results are shown in Table 3. In this experiments, there is no MUX inserted for the simultaneous examination of multiple variables. Instead, each variable is examined one by one. From the table, we can see only small numbers of iterations are required. Comparing to the results in Table 2 with $N = 1$, where any correction is not obtained within 5 hours, the proposed filtering method based on the necessary condition makes the execution time much shorter. It implies that a large number of variables examined in the results shown in Table 2 do not satisfy the necessary condition. The necessary condition works pretty well as filtering.

**Table 3.** Experimental results of filtering candidate variables based on the necessary condition

| | Changed inputs | Time (sec) |
|---|---|---|
| Industrial | 29 | 524 |
| ARM processor | 24 | 293 |

### 4.4 Applying both multiple variable examination and candidate filtering

In the previous experiments, we evaluate our proposed methods for finding variables which can correct circuits when added to inputs of LUTs. That is, simultaneous examination of multiple candidate variables using MUXs and filtering candidate variables based on necessary condition are examined. In this section, we see the effects of applying both of the methods at the same time. For this experiment, we use ISCAS85 circuits and an industrial circuit.

In order to generate buggy designs, one gate in each ISCAS circuit is replaced with an LUT, and one of its inputs is removed from the LUT. As a result, we realize cases where a potentially buggy gate is replaced with an LUT, but it lacks one input for correction because we intentionally remove it. The gate replaced with an LUT and a variable to be removed are randomly chosen, and we make five instances for each ISCAS circuit. For Industrial circuit, we replace one of the buggy gates with an LUT. The replaced LUT needs one more input for correction (without intentionally removing one of its original input) as the original circuit is buggy.

We apply the following three methods for each instance.

- **(PI)** Examining all primary input variables one by one until one can correct the circuit.
- **(Filtering)** Examining only primary input and internal variables one by one which satisfy the necessary condition discussed in Section 3.4.
- **(Filtering + MUX)** Examining multiple variables which satisfy the necessary condition using MUX.

The results are shown in Table 4. In the table, # of var, Corrected, and # of examined represent the total number of candidate variables, (the number of successfully corrected)/(the total number of instances), and the average number of examined variables in successfully corrected cases, respectively. When # of examined is N/A, it means that none of the experimented instances can be corrected by the corresponding method. Ratio means the ratio of the number of examined variables with filtering to the total number of variables. Runtime in the table is the average runtime of the experimented instances.

From the table, we can see the following.

- When we want to correct circuits utilizing programmability of LUT and one additional input to LUT, we need to add some internal variables (not primary input variables) to the LUT.

– When applying the filtering method to filter out variables not satisfying the necessary condition, we can reduce the numbers of examined candidates to 10%-30% of all variables.
– Examining multiple candidates simultaneously using MUXs reduces the runtime significantly.

**Table 4.** Experimental results of applying both of our proposed method

| Circuit | # of var | Method | Corrected | # of examined(ratio) | Runtime(sec) |
|---|---|---|---|---|---|
| c499 | 243 | PI | 0/5 | N/A | 46.4 |
| | | Filtering | 5/5 | 88.6 (36%) | 48.3 |
| | | Filtering + MUX | 5/5 | 88.6 (36%) | 2.3 |
| c880 | 443 | PI | 1/5 | 61.0 (14%) | 80.8 |
| | | Filtering | 5/5 | 54.2 (12%) | 57.0 |
| | | Filtering + MUX | 5/5 | 54.2 (12%) | 2.6 |
| c1355 | 587 | PI | 0/5 | N/A | 60.6 |
| | | Filtering | 5/5 | 155.8 (27%) | 227.7 |
| | | Filtering + MUX | 5/5 | 155.8 (27%) | 3.3 |
| c1908 | 911 | PI | 2/5 | 34.0 (4.0%) | 69.2 |
| | | Filtering | 5/5 | 194.2 (21%) | 284.5 |
| | | Filtering + MUX | 5/5 | 194.2 (21%) | 3.9 |
| c2670 | 1194 | PI | 0/5 | N/A | 708.1 |
| | | Filtering | 5/5 | 142.2 (12%) | 83.2 |
| | | Filtering + MUX | 5/5 | 142.2 (12%) | 4.7 |
| c3540 | 1670 | PI | 0/5 | N/A | 154.9 |
| | | Filtering | 5/5 | 503.8 (30%) | 915.9 |
| | | Filtering + MUX | 5/5 | 503.8 (30%) | 7.8 |
| c5315 | 2476 | PI | 0/5 | N/A | 915.5 |
| | | Filtering | 5/5 | 324.6 (13%) | 268.1 |
| | | Filtering + MUX | 5/5 | 324.6 (13%) | 8.8 |
| c7552 | 3604 | PI | 0/5 | N/A | 1484.3 |
| | | Filtering | 5/5 | 1016.0 (28%) | 3990.1 |
| | | Filtering + MUX | 5/5 | 1016.0 (28%) | 15.9 |
| Industrial | 3209 | PI | 0/1 | N/A | Time out |
| | | Filtering | 1/1 | 100 (3.1%) | 972.3 |
| | | Filtering + MUX | 1/1 | 100 (3.1%) | 172.5 |

### 4.5 Debugging bugs in RTL designs

As a final experiment, the proposed debugging method is applied to the bugs in RTL designs, i.e., incorrect statements in RTL design descriptions, in order to understand how much portions of bugs in RTL designs can be corrected. In general, it is much harder to debug bugs in RTL designs, as multiple portions of the synthesized gate level circuits may have to be corrected even when one statement in RTL is wrong.

We insert one wrong RTL statement into the RTL design descriptions. There are four types of wrong statements we introduce as shown in Table 5. Please note that ”Incorrect variable” and ”Missing variable” most likely need extra inputs to LUTs, and so they are relatively difficult cases. Also ”Incorrect logic” can be difficult cases as the incorrect logic function may make it possible for logic synthesis tools to minimize the circuits incorrectly too much, which results in similar situations as ”Missing variable”.

The results are shown in Table 6. From the results, we can observe the following.

- Bugs in RTL statements are in general more difficult than the ones in gate level designs. Mostly 30-40% cases can be corrected.
- The bugs as ”Extra variable” are relatively easy as expected. This is mostly because we do not need extra variables to be added to LUTs.
- The bugs as ”Incorrect logic” are relatively difficult as they mostly need multiple gates to be corrected.

These observations are important as they suggest for future directions targeting bugs in RTL designs.

**Table 5.** Examples of bugs inserted into ARM processor

| Type of bugs | Correct statement | Buggy statement |
|---|---|---|
| Incorrect logic | assign Z = A & B; | assign Z = A \| B; |
| Extra variable | assign Z = A & B; | assign Z = A & B & C; |
| Incorrect variable | assign Z = A & B; | assign Z = A & C; |
| Missing variable | assign Z = A & B; | assign Z = A; |

**Table 6.** Results on RTL debugging

| Type of bugs | Success ratio | #Added variables | #Candidates | #Iterations | Time (sec) | |
|---|---|---|---|---|---|---|
| | | | | | Success | Fail |
| Incorrect logic | 3/10 | 0 | N/A | 10.4 | 29563.0 | 213.8 |
| Extra variable | 9/10 | 0 | N/A | 16.9 | 38141.4 | 1550.2 |
| Incorrect variable | 3/10 | 3 | 4282.7 | 13.0 | 39192.1 | 173.2 |
| Missing variable | 4/10 | 4 | 5333.8 | 39.0 | 38016.6 | 480.3 |

## 5 Discussions for Extensions

One thing we can observe from the experimental results is that we need sometimes many iterations especially when we need to add additional variables to LUTs. Also, if we can utilize LUTs which have larger numbers of inputs may make the proposed

methods more efficient and more effective. For that direction, one way is, on behalf of LUTs having large numbers of inputs, to introduce fixed topology circuits consisting of a set of LUTs having smaller numbers of inputs, such as the one shown in Figure 12. Here a 12-input circuit is defined with a set of LUTs: two of 4-input/3-output LUTs, three of 4-input/2-output LUTs, and one 4-input/1-output LUT. If we implement 12-input LUT directly, we need $2^{12} = 4K$ bits. On the other hand, the circuit shown in Figure 12 needs only $2^4 \times (2 \times 3 + 3 \times 2 + 1) = 832$ bits, although the number of logic functions that can be realized is much less. The fact that the circuit shown in Figure 12 can only realize very small subset of all possible logic functions with 12-inputs may not be a critical problem if it can represent many or most of the logic functions with 12-inputs actually appearing in real designs.

As a first step of experiments, we apply the proposed debugging methods to synthesize the configuration of the LUTs in the circuit shown in Figure 12 from given specifications. Hence, this corresponds to the identification of logic functions for the LUTs of the 12-inputs circuit, which as a whole can be used in larger circuits. That is, this is an effort to try to accommodate gates with larger number of inputs as candidates of modification when debugging circuits.

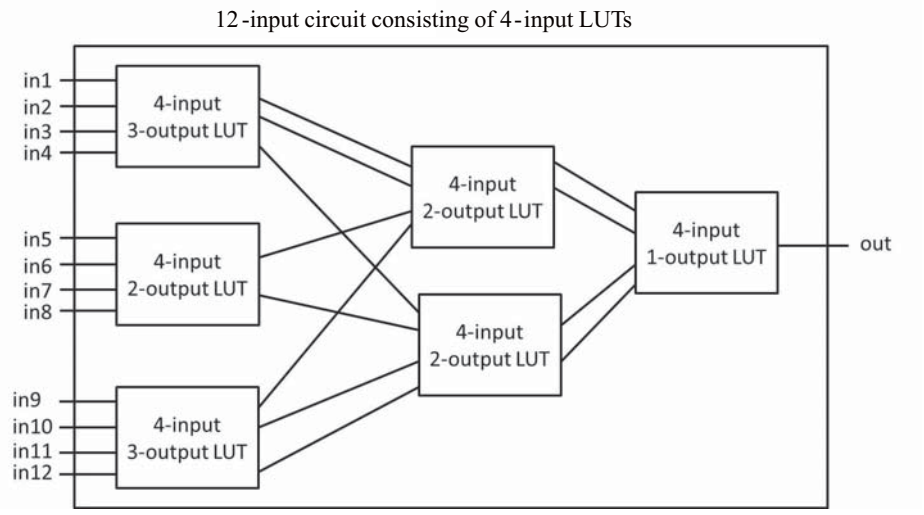


**Fig. 12.** Small LUT-based circuit having 12 inputs

There is a research on enumerating all logic functions appearing in various benchmark circuits [15]. As seen from the chapter, there are not so many logic functions with 12-inputs actually appearing under NPN-equivalence. The chapter also shows the most frequently appearing twenty functions. We have successfully obtained appropriate configurations for the LUTs of the circuit shown in Figure 12 targeting those twenty logic functions. The processing time for one logic function is rather quick, varying from

seconds to hundreds of seconds. Although these results are just for the most frequent twenty logic functions, it is suggesting that the circuit shown in Figure 12 has good flexibility to accommodate various frequently used logic functions and that the proposed debugging methods can really work for the circuits replacing the buggs gates.

## 6 Conclusions and Future Work

In this chapter, we have proposed debugging methods for gate-level circuits applying partial synthesis techniques shown in [2, 7]. In the methods, possible bug locations, which may be given from bug locating methods, are replaced with LUTs, and a configuration of LUTs that makes an implementation under debugging equivalent to its specifications is searched. To deal with the missing input variables to LUTs, we have also proposed methods to examine variables for LUT inputs in trial-and-error manner. Using MUXs, multiple variables are examined simultaneously, which largely reduces the number of iterations of the processes. In addition, we have introduced a necessary condition that variables added to LUT inputs must be satisfied, so that variables not satisfying the condition can be removed quickly from the candidates. Through the experiments with ARM processor design, on-chip network controller taken from industry, and benchmark circuits, we have shown that both of our proposals can significantly speed-up the process to get a correction (i.e. an appropriate configuration of LUTs to make an incorrect implementation correct). We have also shown preliminary experimental results for bugs in RTL designs.

We have also discussed possible extensions of our proposed method, introducing sub-circuits having relatively larger numbers of inputs, such as 12 inputs to the buggy locations of the design under debugging. For such large numbers of inputs, it is not practical to represent the entire sub-circuit with a single 12-input LUT. Instead we have discussed about introduction of decomposition of such sub-circuits with a sets of LUTs having much smaller numbers of inputs.

As a future work, we plan to develop a method to reduce the candidate variables based on the necessary condition discussed in this chapter for the cases where LUTs are dependent with each other. In such cases, the necessary condition may need to be refined to deal with dependency.

## References


1. Yamashita, S., Yoshida, H., Fujita, M.: Increasing Yield Using Partially-Programmable Circuits. In: Proc. of Workshop on Synthesis And System Integration of Mixed Information technologies, pp. 237–242 (2010)
2. Jo, S., Matsumoto, T., Fujita, M.: SAT-Based Automatic Rectification and Debugging of Combinational Circuits with LUT Insertions. In: 2012 IEEE 21st Asian Test Symposium (ATS), pp. 19–24. IEEE Press, New York (2012)
3. Janota, M., Marques-Silva, J.: Abstraction-Based Algorithm for 2QBF. In: Theory and Applications of Satisfiability Testing (SAT) 2011. LNCS, vol. 6695, pp. 230–244. Springer, Heidelberg (2011)

4. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with Counterexample Guided Refinement. In: Theory and Applications of Satisfiability Testing (SAT) 2012. LNCS, vol. 7317, pp. 114–128. Springer, Heidelberg (2012)
5. Ling, A., Singh, P., Brown, S.D.: FPGA Logic Synthesis Using Quantified Boolean Satisfiability. In: Theory and Applications of Satisfiaility Testing. LNCS, vol. 3569, pp. 444–450, Springer, Heidelberg (2005)
6. S.-Lezama, A., Tancau, L., Bodik, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 404–415. ACM, New York (2006)
7. Fujita, M., Jo, S., Ono, S., Matsumoto, T.: Partial synthesis through sampling with and without specification. In: 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 787–794. IEEE Press, New York (2013)
8. Fahim Ali, M., Veneris, A., Smith, A., Safarpour, S., Drechsler, R., Abadir, M.: Debugging sequential circuits using Boolean satisfiability. In: IEEE/ACM International Conference on Computer Aided Design 2004, pp.204–209, IEEE Press, New York (2004)
9. Kosuke Oshima, Takeshi Matsumoto, Masahiro Fujita: A debugging method for gate level circuit designs by introducing programmability. In: IFIP WG10.5 VLSI-SoC, pp.78–83, Oct. 2013.
10. Biere, A.: PicoSAT Essentials. Journal on Satisfiability, Boolean Modeling and Computation (JSAT), vol. 4, pp.75–97 (2008)
11. Brayton, R., Mishchenko, A.: ABC: An Academic Industrial-Strength Verification Tool. In: Computer Aided Verification. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010)
12. Abadir, M. S., Ferguson, J., Kirkland, T.E.: Logic design verification via test generation. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 7, no. 1, pp. 138–148. IEEE Press, New York (1988)
13. AIGER, http://fmv.jku.at/aiger/
14. Icarus Verilog, http://iverilog.icarus.com/
15. Mishchenko, A.,: Enumeration of irredundant circuit structures. In: Proc. of International Workshop on Logic and Synthesis (2014)