Post-Verification Debugging of Hierarchical Designs

Moayad Fahim Ali[1]   Sean Safarpour[1]   Andreas Veneris[1,2]   Magdy S. Abadir[3]   Rolf Drechsler[4]

ABSTRACT

As VLSI designs grow in complexity and size, errors become more frequent and difficult to track. Recent developments have automated most of the verification tasks but debugging still remains a resource-intensive, manually conducted procedure. This paper bridges this gap as it develops robust automated debugging methodologies that complement verification processes. Unlike prior debugging techniques, the proposed one exploits the hierarchical nature of modern designs to improve the performance and quality of debugging. It also formulates the problem in terms of Quantified Boolean Formula Satisfiability to obtain dramatic reduction in memory requirements which allows for debugging of large designs. Extensive experiments conducted on industrial and benchmark designs confirm the efficiency and practicality of the proposed approach.

1. Introduction

As today's VLSI designs grow in complexity and size, design errors become more frequent and difficult to track. These errors are usually caused by an incorrect interpretation of the specification, the human factor, and bugs in CAD tools. Although advancements in verification methodologies have helped reduce the time required to identify an erroneous design [5, 8], engineers are still left with the burdensome task of isolating the source(s) of errors manually. To reduce costs and expedite the debugging effort, efficient automated debugging techniques are necessary to complement and enhance contemporary verification processes [9].

Today, designs are often initially described at high abstraction levels. For example, implementations at the Register Transfer Level (RTL) using Hardware Description Languages (HDL) such as Verilog or VHDL are popular in the industry [14]. Most HDL-based development methodologies make use of hierarchical and modular design concepts to simplify implementation, increase reuse, and facilitate verification [14]. These concepts are especially useful in complex microprocessors, ASICs and Systems-on-Chip that use various Intellectual Property (IP) cores. In this design style, every circuit is a collection of *modules*. Each such module contains a varying number of more basic modules and/or primitive gates. At the lowest level of this hierarchy, a module can contain only primitive logic gates. Most existing debugging techniques do not take the design's hierarchy information into account and operate on a "flattened" gate-level representation. A thorough review of these techniques is presented in [9] and is not discussed here due to space limitations. Since single errors that originate at the RTL level may "translate" into many errors in the flattened gate-level netlist and because debugging is a problem with a complexity that increases exponentially to the number of errors [6], these techniques may not retain a solution due to the increased difficulty of the problem.

Motivated by the above observations, this paper proposes a novel debugging technique for multiple errors in combinational circuits that exploits the design hierarchy. Unlike other approaches, the debugging problem is viewed in terms of erroneous modules rather than erroneous gates. Under this formulation, debugging is performed by traversing the design hierarchy from the highest level to the lowest one while identifying erroneous modules along this path.

Another important characteristic of this method is its use of a Quantified Boolean Formula (QBF) Satisfiability (SAT) solver as the underlying engine that powers debugging. In contrast to other techniques, the QBF formulation avoids any explicit replication of the circuit, providing significant reduction in memory usage. An added benefit is that its performance automatically benefits from rapid advancements in the fields of QBF and SAT solvers [3, 11, 16], especially considering that SAT today provides an attractive proof engine for many formal verification methods [2, 10]. In this respect, the proposed technique encourages further research into QBF evaluation and demonstrates its practicality for logic debugging.

Experiments conducted on a broad range of industrial designs such as microprocessor datapath cores confirm the efficiency and competitive nature of the proposed technique. The hierarchical approach provides a quick way of identifying erroneous modules while the QBF formulation reduces the memory requirements dramatically. As demonstrated empirically, the combination of the hierarchical approach along with the QBF-based operational framework results in a time- and space-efficient debugging framework.

The rest of this paper provides notational conventions and the problem formulation in Section 2. Section 3 presents the basic QBF-based debugging formulation. Section 4 extends this technique to hierarchical designs. Section 5 shows experimental results and Section 6 concludes this work.

2. Preliminaries

In a hierarchical debugging framework, the problem is formulated around the concept of erroneous *design modules* rather than that of erroneous primitive gates [1]. We assume that the combinational gate-level circuit contains n primary inputs $X = \{x_1, \ldots, x_n\}$, m primary outputs $Y = \{y_1, \ldots, y_m\}$ and k internal circuit lines $L = \{l_1, \ldots, l_k\}$. To simplify the presentation, we assume that primary inputs and primary outputs are error-free. We also use the same notation for a line and the QBF variable that corresponds to that line.

A hierarchical design is composed of several interconnected modules, each of which is comprised of smaller submodules and/or primitive gates (glue logic). In this work we treat glue logic as a module composed of only primitive gates. For a given module M_i, $depth(M_i)$ specifies its level in the hierarchy. For instance, the top-level modules have a depth of 1 ($depth(M_i) = 1$) and their submodules have a depth of 2, etc. $SM(M_i)$ refers to the set of M_i's immediate submodules, that is, for any $M_j \in SM(M_i)$, $depth(M_j) = depth(M_i) + 1$. The notation $OUT(M_i)$ refers to the outputs of module M_i, where $OUT(M_i) \subset L$. We call the collection of the above information the *hierarchy information* of the design and we assume that it is known to the algorithm.

Given an erroneous design, its corresponding hierarchy information, a set of counter-examples in terms of q input/output test vectors $T = \{t_1, \ldots, t_q\}$ for which the design fails verification, the approach returns the modules of largest possible depth that may contain errors. This is accomplished by iteratively constructing and solving QBF instances that represent the debugging problem at each hierarchy level. In general, a QBF instance is defined as follows:

$$Q_1 X_1, Q_2 X_2, \ldots, Q_r X_r \ \Phi \qquad (1)$$

[1]University of Toronto, Department of Electrical and Computer Engineering, Toronto, ON M5S 3G4 ({moayad, veneris, sean}@eecg.toronto.edu)

[2]University of Toronto, Department of Computer Science, Toronto, ON M5S 3G4

[3]Freescale Semiconductor, Inc., Austin, TX 78729 (m.abadir@freescale.com)

[4]University of Bremen, Department of Computer Science, 28359 Bremen, Germany (drechsle@informatik.uni-bremen.de)

where Φ is a propositional formula in Conjunctive Normal Form (CNF) over a set of s binary variables $X = \{x_1, \ldots, x_s\}$. The series of Q_i, $i = 1 \ldots r$, is that of an alternating existential ($\exists$) and universal ($\forall$) quantifiers. Every variable from the set X appears in one and only one set X_i, $i = 1 \ldots r$.

3. QBF-based Design Debugging

In this section, we introduce the basics of the QBF-based debugging algorithm. To simplify the discussion, the QBF formulation is first presented in terms of debugging a flat gate-level netlist. In the next section, once the basics have been established, we extend the technique such that it exploits the design hierarchy to handle complex erroneous modules.

To model the debugging problem as a QBF instance, we first add appropriate hardware to the original circuitry to model various constraints of the problem. This enhanced construction, once translated to CNF, represents formula Φ in equation (1). The added hardware contains three components. The first models the ability of different lines in the circuit to correct the design for the test vector set T. The second component provides the input/output test vector constraints. As explained later in this section, this component allows for the memory-efficient QBF formulation of the debugging problem. The third component restricts the number of errors in the design for the purpose of debugging. In the following, each of these components is described in more detail.

The first component models the ability of a candidate error line to correct the circuit for all test vectors in T. To do this, we follow the approach from [6] and introduce a multiplexer with select line s_i on every line l_i ($i = 1 \ldots k$) of the circuit. The multiplexer's 0-input is connected to l_i and its 1-input is connected to a new pseudo primary input w_i. The output of the multiplexer is attached to the original fanout of l_i. We use the notation S to denote the set of all multiplexer select lines $\{s_1, \ldots, s_k\}$. Clearly, when $s_i = 0$ the circuit remains unchanged but when $s_i = 1$ the candidate error line l_i is "disconnected" from the circuit and replaced with the new pseudo primary input w_i. As we will see, if the error on line l_i is excited for test vector t_j, the pseudo primary input w_i will assume the correct/expected logic value of l_i for t_j during QBF-based debugging. A more thorough discussion about the functionality of this hardware in the context of debugging is found in [6].

The second component adds circuitry to the multiplexer-enriched circuit to allow the QBF solver to constrain the design for all test vectors. The objective here is to model input/output constraints that force the erroneous circuit to produce a correct primary output response for every test vector in the counter-example set T. Intuitively, the multiplexer-enriched circuit can produce a correct primary output response only if a subset of multiplexer select lines from S are set to a logic 1, allowing the corresponding pseudo primary inputs to assume logic values that agree with the ones in the correct implementation.

To implement this component, we place a q-to-1 multiplexer at *each* primary input of the circuit and a 1-to-q demultiplexer at *each* primary output where q is the number of test vectors in T. This construction is shown in Fig. 1(a). All of these new multiplexers and demultiplexers share the same $\lceil log(q) \rceil$ set of select lines $V = \{v_1, \ldots, v_{\lceil log(q) \rceil}\}$. When the enhanced circuit is later translated to CNF, logic input (output) values that indicate the *correct* behavior for test vector t_i are placed on the ith input (output) of the appropriate multiplexer (demultiplexer) with the use of unit-literal clauses. Observe, when the select lines of the primary input (output) multiplexer (demultiplexer) assume a binary value of $i - 1$, $1 \leq i \leq q$, then the correct input/output constraints of the ith vector are enforced on the erroneous netlist. For primary input x_i, we use the set $X_i = \{x_i^{t_1}, x_i^{t_2}, \ldots, x_i^{t_q}\}$ to denote the inputs of the respective multiplexer. In this notation, the subscript corresponds to the name of the circuit primary input and the superscript denotes the appropriate test vector. Equivalently, the outputs of the demultiplexer attached at primary output y_i are labeled as $Y_i = \{y_i^{t_1}, y_i^{t_2}, \ldots, y_i^{t_q}\}$.
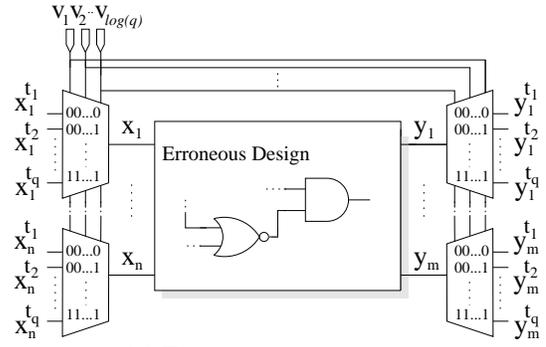
(a) Test vector constraints
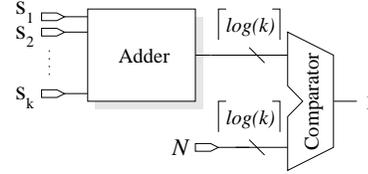
(b) Error cardinality constraints

Figure 1: Second and third hardware components

Example 1: Consider the circuit in Fig. 2(a) and the erroneous implementation in Fig. 2(b) where an inversion is added on line l_1. The counter-example set contains two vectors $T = \{t_1, t_2\}$ where $t_1 = (x_1, x_2, y) = (1, 0, 0)$ and $t_2 = (0, 0, 1)$. In other words, simulation of the correct circuit for t_1 (t_2) yields 1(0) at y.

The first component places a multiplexer on every internal circuit line l_i ($i = 1, 2$). For the second component, a multiplexer is connected to each primary input x_1 and x_2, and a demultiplexer is added to the single primary output y of the circuit. The primary inputs/outputs multiplexers/demultiplexers share the common select line v. The enhanced circuit following the addition of these two components is shown in Fig. 2(c).

When the circuit is later translated in CNF, respective logic values that correspond to test vector t_j, $j \in \{1, 2\}$, are placed on every $x_i^{t_j}$ input of every multiplexer and the $y_i^{t_j}$ output of the demultiplexer using unit-literal clauses. For example, vector t_1 applies logic values 1 and 0 to x_1 and x_2, respectively. To enforce the constraint of t_1 that "simulates" a logic 1 on line $x_1^{t_1}$ of the multiplexer, we apply a unit-literal clause $(x_1^{t_1})$ to the CNF Φ. Similarly, we add a unit literal clause $(\overline{x_2^{t_1}})$ to represent that input $x_2^{t_1}$ "simulates" a logic 0 and finally a unit literal clause (y^{t_1}) to indicate that the correct simulation value (of the correct implementation) of y for t_1 is 1. We let the reader verify that the required unit-literal clauses for all test vectors in T are $(x_1^{t_1})(\overline{x_2^{t_1}})(y^{t_1})(\overline{x_1^{t_2}})(\overline{x_2^{t_2}})(y^{t_2})$. Observe in Fig. 2(c), when the value of the select line $v = 0(1)$, test vector $t_1(t_2)$ constrains the (erroneous) enhanced circuit to the *correct* primary input/output values for that vector. ■

The third component restricts the number of select lines that can be set to a logic 1 simultaneously. Intuitively, this component indicates that exactly N errors are present in the circuit. Since N is unknown, the debugger will first try to return a solution for $N = 1$. If it fails, the value of N is incremented ($N = 2$), etc, until a solution is found. This process is described in greater detail in Section 4. These requirements are coded using the *error cardinality* circuitry from [6]. That circuitry performs a bitwise addition of the select lines in S and forces the sum to be equal to N, as shown in Fig. 1(b). Its effects, when translated in CNF, is that when more *or* less than N select lines are set to 1, it automatically causes the formula to become unsatisfiable and the solver to backtrack. This component is not depicted in the figures and examples that follow for easier

(a) Error–free design   (b) Erroneous design

(c) QBF–based debugging
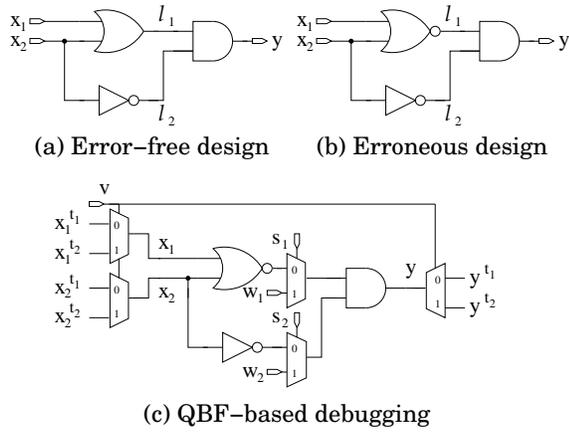
Figure 2: Circuit for Examples 1 and 2

illustration.

Once all three hardware components are added, the enhanced circuit is translated to CNF which is then padded with input/output values of the test vectors in the form of unit-literal clauses, as demonstrated in Example 1. Once formula Φ is available, the problem of debugging is equivalent to the following statement: *Does there exist a set of exactly N select lines $s_{i_1}, s_{i_2}, \ldots, s_{i_N}$ that can be set to a logic 1 such that for each input test vector there exist logic values for pseudo primary inputs $w_{i_1}, w_{i_2}, \ldots, w_{i_N}$ that justify the (correct) primary output behavior of the circuit?* Formally, this statement is expressed by the following QBF problem:

$$\exists\, s_1, \ldots, s_k\ \forall\, v_1, \ldots, v_{\lceil log(q) \rceil}\ \Phi \qquad (2)$$

In the above formula, any variables not explicitly included in the first level ($\exists$) or second level ($\forall$) quantifier are implicitly included in a third level ($\exists$) quantifier. Any satisfying assignment to this problem will contain exactly N select lines $s_{i_1}, s_{i_2}, \ldots, s_{i_N}$ set to a logic 1, corresponding to the candidate error gates. If no such satisfying assignment exists, the design contains more than N errors and the debugger increments the value of N to solve the QBF again. When the algorithm fails for a particular value of N, the QBF instance can be reused by simply changing the unit-literal clauses that enforce the value of N.

Observe, under this problem formulation, the set of select lines that are set to a logic 1 must be common to all test vectors. However, logic values on the respective $w_{i_1}, w_{i_2}, \ldots, w_{i_N}$ multiplexer inputs are allowed to be different for each test vector since the variables corresponding to these lines are at a higher quantification level than of those that correspond to V. In modern QBF solvers, these values can be maintained for each test vector to aid the correction effort [6].

Example 2: To generate the QBF instance for the enhanced circuit in Example 1, we first add the error cardinality hardware to the enhanced circuit shown in Fig. 2(c) and we translate the new circuitry to CNF. Then, input/output $\{t_1, t_2\}$ test vector behavior and $N = 1$ are enforced with unit-literal clauses to get the final Φ. The debugging problem can now be expressed as

$$\exists\, s_1, s_2\ \forall\, v\ \Phi$$

where a QBF solver returns the solution $\{s_1, s_2\} = \{1, 0\}$ to indicate that line l_1 is a candidate error line. ∎

This QBF-based debugging methodology can be extended to handle sequential circuits by applying a similar technique to the one in [6]. However, sequential logic debugging is not the topic of this work and is not addressed here.

3.1   Analysis and Comparison

The memory requirements for the QBF instance of the debugging problem as expressed in equation (2) are $O(k + q(n + m))$ where k

is the number of circuit lines, n is the number of primary inputs, m is the number of primary outputs and q is the number of test vectors. To see this, the multiplexer-enriched circuit and the error cardinality hardware require $O(k)$ number of clauses [6]. For each primary input, the q-to-1 multiplexer can be implemented using at most number $q - 1$ of 2-to-1 multiplexers organized in a binary tree fashion. Since a 2-to-1 multiplexer requires four CNF clauses, the multiplexers that express the input test vector constraints require $O(qn)$ clauses. Similarly, the demultiplexers that enforce the output test vector constraints can be implemented with $O(qm)$ clauses.

The above analysis shows why our methodology is space-efficient when contrasted to existing SAT-based debugging approaches, more notably the one in [6]. That Boolean SAT debugging methodology "duplicates" the circuit for each input test vector for a total of $O(q(k + n + m))$ clauses. Since the number of lines k is the dominating factor and because the debugging resolution depends on the number q of test vectors used [9, 6], that approach trades memory for run-time performance in large designs. The QBF modeling of the debugging problem provides a time/space trade-off, reducing the memory requirements through universal quantification of the test vectors. In other words, the proposed QBF problem representation allows the solver to explore all vector constraints "on the fly", while solving the formula, and it does not require these constraints to be enforced explicitly, that is, when the formula is built.

Another major advantage of the QBF debugging formulation is that conflict clauses found for one test vector are automatically reused for all other test vectors as well. Conflict clauses often improve the performance of SAT and QBF solvers by pruning sections of non-solution space [12, 16]. When a circuit is replicated for each test vector, conflict clauses cannot be reused since each instance of the circuit contains different CNF variables for the same circuit line.

4.   Hierarchical Debugging

Modern designs are usually captured in a hierarchical and modular manner. Once a design fails verification, information about the erroneous modules is of great interest to the designer. This is partly because automated synthesis environments do not require the designers to work on the gate-level. As a result, few designers are familiar with the gate-level details of the design and interaction with higher-level representations is preferred.

Another reason why debugging flattened gate-level netlists may not be ideal is because simple errors introduced at higher abstraction levels of the design flow may manifest themselves into numerous gate-level errors. A debugging technique that targets simple gate primitives may have trouble identifying all such erroneous gates. This is due to the large search space that grows exponentially to the number of erroneous gates [6]. Debugging for erroneous modules, where each module may contain many erroneous gates, does not only reduce the problem complexity but also provides more aid to the engineer through better error localization.

In light of the above, this section generalizes the gate-level debugging methodology from Section 3 to hierarchical designs. The proposed approach exploits the design hierarchy and operates in a *top-down* debugging fashion. Starting at the top-level ($depth = 1$) of the hierarchy and progressing towards the lowest levels, erroneous modules are examined in an iterative manner along this path.

Clearly, if the effects of an error within module M_i propagate to the primary outputs, then they also propagate to one or more outputs in $OUT(M_i)$ of module M_i. Consequently, instead of using the multiplexers in the first component (Section 3) to model potential error sites on every circuit line, we can use them to model potential erroneous modules at all output lines $l_o \in OUT(M_i)$ of module M_i. Under this scenario, all multiplexers placed on the outputs of a module M_i must share a common select line s_{M_i} to indicate the potential of this individual module to correct the design.

The iterative algorithm starts by first placing multiplexers at the outputs of all top-level modules M_i such that $depth(M_i) = 1$. The QBF instance of the debugging problem is then formed as described in Section 3. A solution to this instance results in a set of erroneous modules denoted as M_{err}^1. In the next iteration, multiplexers are

(a) Two-bit full-adder

(b) One-bit full-adder   (c) Half-adder

(d) Debugging in first iteration

(e) Debugging in second iteration
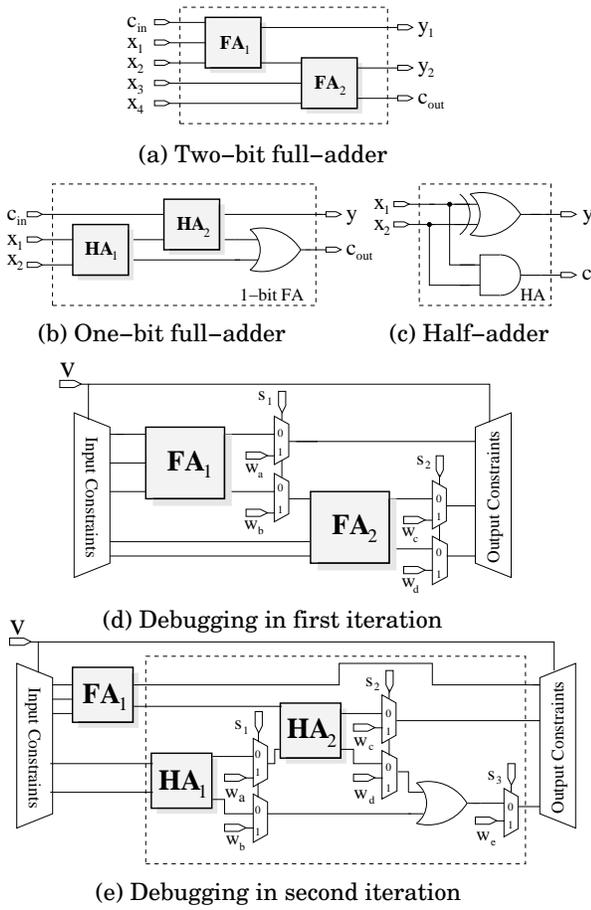
Figure 3: Hierarchical debugging of a full-adder

placed at the outputs of all modules M_j such that $depth(M_j) = 2$, $M_j \in SM(M_i)$, and $M_i \in M_{err}^1$ to form a new QBF instance and have the solver search for candidate error modules in the second hierarchy level. This process is repeated until the lowest hierarchy level when an erroneous module (for fixed value of N) is reached.

In the first iteration, the QBF instance is solved with $N = 1$. If in the ith iteration the solver fails to return a set of candidate erroneous modules, N is incremented and the ith iteration is repeated. For the $(i+1)^{th}$ iteration, N is not reset to 1 but it maintains its value from the ith iteration because the solver has already deduced that *at least* N erroneous modules exist in the circuit. Example 3 demonstrates this hierarchical debugging methodology.

Example 3: Consider the hierarchical design of a two-bit full-adder in Fig. 3(a). This full-adder is comprised of two one-bit full-adders (Fig. 3(b)) each of which contains two half-adders shown in Fig. 3(c). Assume that the design error is an added inversion on the AND gate in the first half-adder of FA_2. A verification tool provides the counter-example set $T = \{t_1, t_2\}$ where $t_1 = (c_i, x_1, x_2, x_3, x_4, y_1, y_2, c_{out}) = (00011000)$ and $t_2 = (00100101)$.

Fig. 3 (d) illustrates the hardware construction during the first hierarchical debugging iteration when the depth of the error modules explored is 1. Note that all multiplexers placed at the output lines of each full-adder share a common select line. Let Φ_1 denote the corresponding CNF for this hardware, including the error cardinality circuitry (not shown here), and the set of unit-literal clauses used to apply the test vector constraints for the second component. Then, the QBF problem for this iteration is stated as follows:

$$\exists\{s_1, s_2\} \, \forall\{v\} \, \Phi_1$$

For $N = 1$, the solver returns the solution $\{s_1, s_2\} = \{0, 1\}$, indicating that the set of candidate erroneous modules at this hierarchy level is $M_{err}^1 = \{FA_2\}$.

In the second iteration, multiplexers are placed at the outputs of the half adders HA_1 and HA_2 of the candidate error module FA_2 and that of the OR gate (glue logic) since a gate, by default, is the simplest instance of a module (recall that we treat glue logic as a module comprised of primitive gates). Each set of these multiplexers share a common select line. No multiplexers are placed at the outputs of FA_1 or any of its submodules since it has been recognized as error-free during the first iteration. Fig. 3(e) illustrates the construction behind this iteration where the resulting QBF instance is:

$$\exists\{s_1, s_2, s_3\} \, \forall\{v\} \, \Phi_2$$

where Φ_2 is the CNF formula that corresponds to the circuit in Fig. 3(e). This QBF instance has two distinct solutions. The first is $\{s_1, s_2, s_3\} = \{1, 0, 0\}$ to indicate that an error exists in module HA_1, while the other is $\{s_1, s_2, s_3\} = \{0, 0, 1\}$ to suggest an error at the glue logic of module FA_2. At this point, the hierarchical debugging algorithm terminates with the above solution since no module in M_{err}^2 contains any submodules. However, it is important to note that this technique is flexible and can obtain improved resolution by debugging one level of hierarchy deeper. In an optional final iteration, the algorithm considers each primitive gate in the erroneous HA_1 as a single output module. This identifies the AND gate as the source of the error. ∎

Algorithms 1 and 2 illustrate the main procedures of the QBF hierarchical debugging methodology. QBF_HIER_DEBUG compiles the set of modules to be debugged at each iteration. This set is then passed to DEBUG_ITER for the actual debugging. The input to QBF_HIER_DEBUG is the circuit C, the set of top level modules M_{top}, the maximum number of errors $maxN$ to search for, and the counter-example set T. Line 4 initializes the current set of modules to be debugged, M_{cur_iter}, for the first iteration. The while loop in line 5 is executed once per debugging iteration and $flag$ becomes false only when the lowest level of the hierarchy is reached.

The loop in lines 7-12 executes repeatedly as long as procedure DEBUG_ITER fails to return a solution for the current value of N, upon which N is incremented. When N exceeds the user defined $maxN$, the procedure terminates with no solution. In lines 15-20, the set of modules to be debugged in the next iteration, M_{nxt_iter}, is compiled based on M_{err} of the current iteration. In the case that erroneous module $M_i \in M_{err}$ contains submodules, $flag$ is set to true and M_i's submodules are added to M_{nxt_iter}. Otherwise, M_i itself is added to M_{nxt_iter}. This is necessary when debugging for multiple erroneous modules.

Algorithm 2 illustrates the procedure DEBUG_ITER which performs debugging for one iteration. Lines 3-11 convert the enhanced circuit and compile the QBF instance as explained earlier in this paper. In line 12, the formula is passed to the QBF solver which seeks all solutions. If a satisfying solution is found, the algorithm identifies the active select lines (or modules) and adds a clause that "blocks" this solution as in [6]. This provides an *all-solution* debugging engine that reuses past computation for efficiency. The modules corresponding to the active select lines are also added to the set of erroneous modules. This process is repeated to identify all possible solutions, until the QBF solver returns with no solution.

4.1 Multiple Erroneous Modules

In the case of multiple erroneous modules, a situation may arise where error effects from one module *mask* the error effects of another. This may result in the solver identifying only a subset of the failing modules. Consider Fig. 4, for example, where modules Q and R are erroneous and assume that $N = 1$. In the first iteration, B is identified as a single-module solution because $OUT(B)$ dominates $OUT(R)$ (and $OUT(Q)$). Since a solution was found, N will not be incremented and the solver will move to the next hierarchy level by attaching multiplexers only on all submodules of B but not

Algorithm 1 QBF-based Hierarchical Debugging

```
 1  QBF_HIER_DEBUG(C, M_top, maxN, T)
 2  begin
 3      N := 1
 4      M_cur_iter := M_top
 5      while (flag = true)
 6          flag := false
 7          while (M_err = ∅)
 8              M_err = DEBUG_ITER(C, M_cur_iter, N, T)
 9              if (M_err = ∅)
10                  N := N + 1
11                  if (N > maxN)
12                      return ∅
13
14          foreach (M_i ∈ M_err)
15              if (SM(M_i) != ∅)
16                  M_nxt_iter := M_nxt_iter ∪ SM(M_i)
17                  flag := true
18              if (SM(M_i) = ∅)
19                  M_nxt_iter := M_nxt_iter ∪ M_i
20          M_cur_iter := M_nxt_iter
21      return M_err // At this point, M_err = M_cur_iter
22  end
```

Algorithm 2 Hierarchical Debugging of a Single Iteration

```
 1  DEBUG_ITER(C, M_cur_iter, N, T)
 2  begin
 3      foreach (M_i ∈ M_cur_iter)
 4          associate select line s_i with M_i
 5          foreach (l_j ∈ OUT(M_i)
 6              insert MUX on l_j with select line s_i
 7      generate second hardware component
 8      generate third hardware component
 9      translate structure into CNF instance Φ
10      apply test vectors using unit-literal clauses
11      generate ∃ and ∀ quantifiers and QBF instance
12      while (qbf_solve() = SAT)
13          let sol : Set of active select lines
14          let c : Clause := ∅
15          foreach (s_i ∈ sol)
16              c := c + s̄
17          Φ := Φ · c // Disable sol so that it is not found again
19          foreach (s_i ∈ sol)
20              let M_i : module associated with s_i
21              M_err := M_err ∪ M_i
22      return M_err // Erroneous modules
23  end
```

the ones of A. In this case, the solver will not be able to satisfy the problem and it will return with failure.

To tackle this problem, a preprocessing step is required before debugging can commence. For each erroneous primary output y_i, we compile the set of modules $FI(y_i)$ in the fan-in cone of y_i. Using this information, in the next iteration the algorithm does not only debug the contents of modules in M_{err} but it also examines the contents of those modules that appear with any $M_j \in M_{err}$ in the same $FI(y_i)$ for any erroneous primary output y_i. Clearly, those modules must also have the same depth as module M_j. Revisiting Fig. 4, in the first iteration we debug A, B, and C. When B is returned by the solver, R and S (along with Q) are automatically included in the next iteration since A and B lie in the same fan-in cone of erroneous primary output y_1. The module masking information can be obtained once in a fast, linear-time, preprocessing step. This step guarantees the completeness of the debugging process where all possible erroneous modules will be identified for a given set of counter-examples regardless of the effects of error masking.
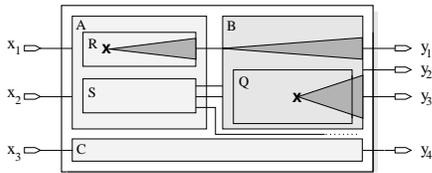
Figure 4: Error masking in multiple erroneous modules

5. Empirical Results

This section presents empirical results from a broad range of industrial designs. Table 1 contains information about the circuit characteristics. In this table, the first and second columns show the design name and description, respectively. The total number of synthesized gates in the circuit is shown in column three. Column four presents the total number of design modules accumulated over all levels of the hierarchy, while the fifth column shows the maximum depth of the design hierarchy. Circuits c3, c5, c6, and c7 are industrial commercial microprocessor datapath core designs, while the remaining circuits are available as open source cores/designs [13].

The QBF-based hierarchical debugging algorithm is implemented in C++ using the QBF solver Quaffle as the underlying engine [16]. The technique is independent of the QBF solver type and other DPLL or resolution-based solvers [4, 7] can be utilized. The experiments are run under a Linux platform on an Intel Pentium 2.7GHz workstation with 1GB of RAM. For each circuit, ten experiments are conducted with the reported results averaged out. For each such experiment, thirty counter-examples are used and run-times are reported in seconds.

Two sets of different experiments are performed. In the first set, the interconnection of one module with the design is randomly perturbed to generate an error. This may result in more than one error between the erroneous module and other modules. Such type of errors are common when system integrators are interconnecting embedded cores in large SoCs [15]. In the second set, we arbitrarily change the functionality of one, two or three modules at the RTL level. When the design is later synthesized, these errors usually translate into many erroneous primitive gates in the final netlist. In both cases, the errors are inserted at deep hierarchy levels which makes the debugging process harder.

Results for the first set of experiments for module interconnection errors are shown in Table 1. Column six contains the average depth of the erroneous module found. Column seven shows the total runtime in seconds which includes all module-level debugging iterations but excluding the final gate-level iteration discussed in Example 3. We observe that the runtime increases with deeper erroneous modules because more debugging iterations are required before a solution is identified. In column eight, the average number of clauses in the compiled QBF is shown, while column nine shows the memory requirements in MB. In all experiment, the method returns with the erroneous module. From these results it is clear that the memory requirement remains small even for large designs.

Table 2 presents experiments for the second set of experiments that relates to functional changes in random modules at the HDL level. Again, the approach is able to return the erroneous modules in all cases. Debugging results are presented in a similar manner as in Table 1 with the exception of columns 2, 7 and 12, which show the average number of primitive gate errors resulting from errors introduced at the RTL. These numbers demonstrate the practicality of debugging approaches that can handle hierarchy information. This information helps the designer localize the source of error irrespective of the number of erroneous primitive gates which may be large and misguiding to a traditional debugging approach. In addition, Table 2 shows that the proposed technique scales well for $N > 1$. This is due to the iterative debugging methodology which prunes large sections of the nonsolution space in each step, leaving fewer modules to be examined in the subsequent iterations.

Table 1: QBF-based hierarchical debugging results for module interconnection errors

Ckt	Circuit Description	# of gates	# of mods	Max. depth	Avg. depth	Time (sec)	# of clauses	Mem. (MB)
c1	Reaction timer circuitry	278	8	3	2.0	0.05	3,589	18.6
c2	Fibonacci number generator	723	69	3	2.8	0.12	5,396	19.0
c3	Microprocessor Datapath	2,159	11	3	2.7	0.85	29,737	22.2
c4	8-bit RISC Processor	3,141	21	3	2.0	0.65	35,465	26.8
c5	Microprocessor Datapath	5,484	50	5	4.4	3.56	62,319	28.7
c6	Microprocessor Datapath	10,205	46	4	3.2	2.88	102,401	33.5
c7	Microprocessor Datapath	11,265	155	4	3.1	2.61	71,754	33.2
c8	32-bit Pipelined RISC Processor	25,262	30	4	2.2	2.56	168,450	57.2
Avg.		7,360	48.8	3.6	2.8	1.66	59,888	29.9

Table 2: QBF-based hierarchical debugging results for module design errors

Ckt	Single Erroneous Module ($N=1$)					Double Erroneous Modules ($N=2$)					Triple Erroneous Modules ($N=3$)				
	# of errors	Avg. depth	Time (sec)	# of clauses	Mem. (MB)	# of errors	Avg. depth	Time (sec)	# of clauses	Mem. (MB)	# of errors	Avg. depth	Time (sec)	# of clauses	Mem. (MB)
c1	2.6	2.1	0.06	3,132	18.6	8.7	2.0	0.07	3,868	19.2	12.0	2.2	0.08	3,952	19.8
c2	2.9	2.9	0.13	5,396	19.8	5.7	2.9	0.28	5,627	19.9	9.0	2.0	0.40	5,544	19.0
c3	6.9	2.9	0.69	29,573	22.3	12.2	2.0	0.93	31,993	22.3	40.0	2.1	0.94	30,441	22.2
c4	6.4	2.2	0.76	40,096	23.1	10.8	2.1	1.56	41,078	26.2	40.3	2.0	1.95	38,055	26.3
c5	5.4	4.1	3.08	63,805	28.7	14.4	3.1	4.02	65,750	28.7	35.5	4.0	5.30	69,972	28.9
c6	6.2	3.1	2.64	103,201	33.5	10.8	3.3	3.46	104,476	33.5	39.8	2.0	4.05	106,799	33.6
c7	6.5	3.0	2.66	71,766	32.9	13.6	3.1	3.21	72,449	33.1	42.0	3.0	3.62	74,625	33.2
c8	3.0	3.0	4.63	167,198	57.2	14.0	2.0	3.62	172,195	57.2	50.0	2.3	4.57	172,195	57.3
Avg.	5.0	2.9	1.83	60,520	29.5	11.3	2.6	2.14	62,179	30.0	33.6	2.45	2.61	62,697	30.0
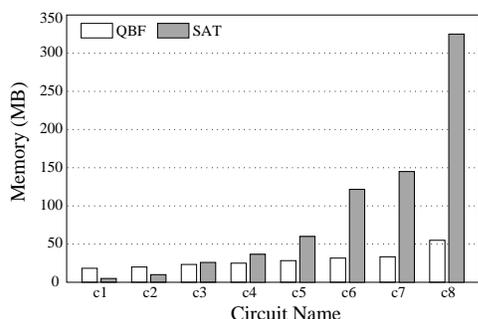
Figure 5: Memory requirements

To analyze the memory benefits from the QBF formulation, Fig. 5 depicts a comparison with a SAT-based flat gate-level netlist debugging technique that replicates the circuit for each test vector. The memory requirements are shown for experiments when a single module is corrupt. For the two smallest circuits, the QBF formulation requires slightly more memory due to the dominant size of the test vector constraint hardware when $q = 30$. However, for all other circuits where the size of the circuit dominates the problem, the QBF formulation provides a clear advantage with up to 5.7 times reduction in memory. This confirms the linear memory requirements of the approach as shown in subsection 3.1. It also demonstrates the viability of the approach in a real-life industrial environment. It should be noted, memory requirements of the method for higher values of N are very similar to those in Fig. 5, as Tables 1 and 2 and the analysis in subsection 3.1 also suggest.

6. Conclusion

Although most verification tasks have been automated, debugging remains a manual resource-intensive procedure. This paper proposes a debugging technique for multiple design errors in combinational circuits. It reduces the debugging problem into an instance of QBF satisfiability where solvers can be utilized to guarantee performance yet minimize memory requirements. Another advantage is that it uses the design's hierarchy information to expedite the process of debugging. A comprehensive suite of experiments on real-life designs confirm its effectiveness as a complementary process to formal verification and promotes further work in the field.

7. References

[1] M. S. Abadir, J. Ferguson and T. E. Kirkland, "Logic Verification Via Test Generation", in *IEEE Trans. on CAD*, vol. 7, pp. 138–148, Jan. 1988.

[2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman and Y. Zhu. "Bounded Model Checking", *Vol. 58 Advances in Computers,* Academic Press, 2003.

[3] D. Chai and A. Kuehlmann, "Fast pseudo-Boolean constraint solver", in *Proc. of DAC, pp. 830–835,* June 2003.

[4] K. Chandrasekar and M.S. Hsiao, "Q-PREZ: QBF Evaluation Using Partition, Resolution and Elimination with ZBDDs," in *Proc. of VLSI Design, pp. 189–194,* Jan. 2005.

[5] R. Drechsler, *Advanced Formal Verification*, Kluwer Academic Publishers, 2004.

[6] M. Fahim Ali, A. Veneris, S. A. Safarpour, R. Drechsler, A. Smith, and M. Abadir, "Debugging sequential circuits using Boolean satisfiability," in *Proc. of ICCAD, pp. 204–209,* Nov. 2004.

[7] E. Giunchiglia, M. Narizzano, and A. Tacchella, "QuBE++: An Efficient QBF Solver," in *Proc. of FMCAD, pp. 201–213,* Nov. 2004.

[8] G. Hachtel and F. Somenzi, '*Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, 2000.

[9] S. Y. Huang and K. T. Cheng, *Formal Equivalence Checking and Design Debugging*, Kluwer Academic Publishers, 1998.

[10] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," in *IEEE Trans. on CAD, vol. 11, no. 1, pp. 4-15,* Jan. 1992.

[11] F. Lu, L.-C. Wang, K.-T. Cheng and R. Y.-Y. Huang, "A Circuit SAT Solver with Signal Correlation Guided Learning," in *Proc. of IEEE DATE, pp. 892-897,* 2003.

[12] M.H. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. of DAC, pp. 530-535,* June 2001.

[13] OpenCores.org, *http://www.opencores.org/*.

[14] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective*, Prentice Hall, 1996.

[15] C.-Y. Wang, S.-W. Tung and J.-Y. Jou, "Automatic Interconnection Rectification for SoC Design Based on the Port Order Fault Model," in *IEEE Trans. on CAD, vol. 22, no. 1, pp. 104-114,* Jan. 2003.

[16] L. Zhang and S. Malik, "Conflict Driven Learning in a Quantified Boolean Satisfiability Solver," in *Proc. of ICCAD, pp. 442–449,* Nov. 2002.