Error Detection Using Gate-Level Assertions

Daniel U. Becker, Hakan Baba
{dub, hakan}@stanford.edu

***Abstract*—The following work investigates the use of hardware assertion-checkers based on gate-level assertions as a mechanism for error detection and built-in self test (BIST). We introduce a tool flow for automatically generating appropriate checker modules from a netlist, and investigate the tradeoff between the fault coverage that can be achieved by these checkers and the hardware overhead that is associated with them.**

I. Introduction

Hardware verification aims to ensure that a chip complies with given specifications. To ensure correct operation of a VLSI chip, several measures must be taken throughout the production process. For example, both functional and electrical (e.g. margin violations) errors can be introduced during the design phase; similarly, impurities and statistical variations during the production process can also result in incorrect circuit operation. In order to identify erroneous circuits, verification can be used both at design time and, with suitable hardware support, after fabrication.

While historically, assertions have primarily been used in the context of verification to find functional errors during the design phase, recent work has proposed the inclusion of hardware assertion checkers to enable post-silicon failure diagnosis and to provide BIST functionality in the field.

In the present work, we explore the use of gate-level assertion checkers for online circuit verification and try to evaluate the associated cost-complexity tradeoff.

II. Background and Related Work

High-level assertions are commonly employed during the design process to find and eliminate implementation errors and deviations from the specification. Such assertions are typically manually generated by the designer or verification engineer based on intuition or English-language specifications, and aim to capture the functional design intent. A circuit's correct operation is verified by either using formal techniques to prove that the assertions always hold, or by applying sufficiently large sets of input patterns in simulation until the desired level of confidence is reached.

More recently, the use of assertions has been extended beyond the design phase through the inclusion of dedicated hardware-based assertion checkers to the design. These hardware assertion checkers can be used to facilitate failure diagnosis during post-silicon debug, and can also provide BIST capabilities in the field.

In addition to the high-level assertions typically used in the context of verification, the design space for such hardware checkers also includes low-level assertions that are specified at the gate level. Such gate-level assertions capture implications between different nets in the design; they can be automatically extracted from a design's netlist, and are typically neither visible nor useful during the design phase. Gate-level assertions are oblivious to a design's high-level functionality. Instead, they verify its physical characteristics, such as the absence of stuck-at faults, timing errors, electrical violations, etc.

In an assertion based self test architecture, input patterns are applied to the design under test and assertion checker hardware is used for output response analysis. Consequently, the output of the self test is not a signature, to be compared with a previously stored one that characterizes a properly operating circuit; instead, the output comprises signals that directly indicate either correct operation or circuit failure. This leads to a significant reduction in test data volume. Furthermore, assertion checker circuits can easily be extended to provide additional information that helps to narrow down the time and location in the design at which the problem occurred, thereby simplifying the debug process for complex digital circuits both before and after first silicon is available.

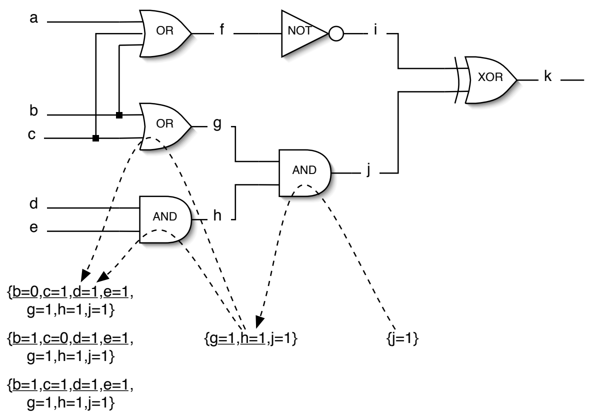
Previous work has investigated the use of implications for search space pruning in the context of ATPG algorithms: implications can assist in recognizing redundant and untestable faults during test pattern generation. [1] and [2] propose efficient techniques for extracting implications from a circuit. However, their primary focus is on the optimization of test pattern generation, whereas we use implications as the basis for hardware assertion checkers for BIST.

Seshia et al [3] have proposed the use of high-level assertions in conjunction with formal verification for identifying which latches in a circuit must be protected against soft errors in order to ensure that the circuit complies with formal specifications.

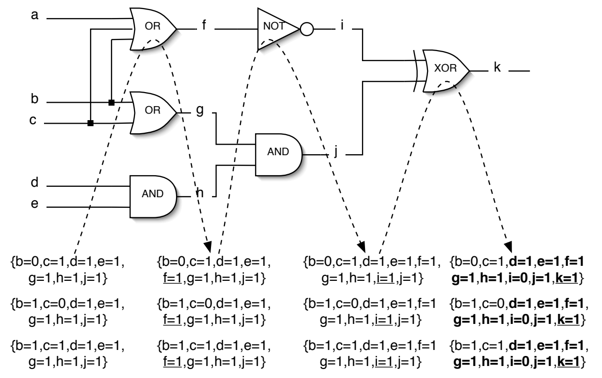
Finally, recent research has investigated the inclusion of checker modules based on high-level assertions in hardware emulators and in shipping hardware in order to provide extended BIST and in-field diagnosis capabilities. Boulé et al [4] employ MBAC—a tool developed at McGill University [5], [6], [7]—to generate assertion-based hardware checkers for an AMBA slave device and an AMBA AHB bus interface, and analyze the associated area overhead and cycle time penalties. While MBAC automatically generates hardware checker modules from assertions specified in an assertion language such as SystemVerilog Assertions (SVA) or Property Specification Language (PSL), the assertions themselves have to be created manually by the designer. In contrast, our evaluation considers automatically generated gate-level assertions.

III. Implementation and Methodology

We have developed a tool flow that automatically generates a set of hardware-based, gate-level assertion checkers for a



(a) Backward propagation



(b) Forward propagation

Fig. 1: Logic value propagation example

given circuit. Assertions are selected based on fault coverage, and the user is presented with a tradeoff between the achievable fault coverage and the associated cost in terms of hardware overhead.

In particular, our approach comprises the following steps:

A. Assertion Extraction

In an initial step, we perform a full implication analysis on the circuit under consideration. In order to find gate-level implications in the design, we use a two-step logic value propagation process. Figure 1 shows shows an example of the logic value propagation steps for a simple circuit.

We begin with the initial value assignment $j = 1$ in In figure 1(a) and traverse the network towards its inputs. Because j is the output of an AND gate, $j = 1$ implies that the gate's inputs, g and h, also be high, resulting in a new state $\{g = 1, h = 1, j = 1\}$. Similarly, $h = 1$ implies $d = 1$ and $e = 1$. The situation is slightly more complicated for b and c: because $g = 1$, we know that at least one of these two nets must also be high. However, without additional information about the circuit, we cannot determine which combination of b and c should be chosen; therefore, we have to expand out state space to include three valid states as shown in figure 1(a). We continue our backward propagation in the same fashion until we either reach the network's primary inputs or, if defined, a recursion limit. In particular, we can terminate recursion after reaching a given number of states, after traversing a given number of gates, or any combination of the two; our results indicate that relatively small state count thresholds are sufficient to find almost all implications while keeping CPU time and memory requirements at reasonable levels.

Backward propagation yields a set of valid network states that are compatible with the initial value assignment. For each generated starting state, we then proceed to perform a full logic simulation as shown in figure 1(b). States are augmented whenever simulation yields a new logic value assignment, and discarded in case a conflict is detected. Nets that cannot be fully evaluated—e.g. as a result of a sensitized gate input with unknown value—are simply ignored.

At the end of forward propagation, we have a set of possible network states that are compatible with our initial assignment. To identify implications, we must now find logic value assignments that are common across all generated states. Looking at figure 1(b), we find that the assignments printed in bold satisfy this requirement. Thus, in the given example, the initial assignment $j = 1$ has the following implications: $d = 1$, $e = 1$, $f = 1$, $g = 1$, $h = 1$, $i = 0$ and $k = 1$.

We have written a command-line tool that parses a Verilog netlist, performs a complete implication analysis for both possible logic values of each net contained in the design, filters out trivial (e.g. $a = 1 \implies a = 1$) and redundant (e.g. $a = 1 \implies b = 1$ and $b = 0 \implies a = 0$) assertions, and generates a set of augmented netlists that contain a single hardware assertion checker each. These files serve as the inputs to the next stage.

B. Fault Simulation

In order to be able to make informed decisions about which set of assertions to pick for inclusion in hardware, we need to generate fault coverage information for each individual assertion. We chose to evaluate fault coverage using the TetraMax NG fault simulator by Synopsys.

For each assertion that was generated in the previous step, we create an augmented netlist which contains the original netlist and the hardware checker module corresponding to the assertion. The checker modules consist of a basic two-input gate with one of its inputs potentially inverted[1]. To ensure that we only test fault coverage that is due to the checker in question, all circuit outputs with the exception of the checker's are removed from the augmented netlist. Furthermore, TetraMax NG is configured to only account for faults that were also present in the original netlist; i.e., we ignore faults within the checker logic.

After performing a fault simulation run for each assertion in this way, we have a set of files that indicate which faults each assertion checker covers individually, which are then processed in the final step.

C. Assertion Selection

The last step in our algorithm is the selection of a minimum subset of assertions that yields the desired fault coverage.

[1]The exact gate required can be easily determined by noting that the implication $a = 1 \implies b = 1$ is equivalent to the expression $\neg a \lor b$ and inverting inputs as appropriate.

Circuit	# Assertions (CPU time) 8 state limit		1024 state limit	
eth_clockgen_DW01_dec_1	224	(0.016s)	224	(0.083s)
eth_txcounters_DW01_cmp2_0	323	(0.032s)	321	(15.714s)
eth_txcounters_DW01_dec_1	2849	(0.111s)	2849	(43.591s)
eth_txcounters_DW01_sub_1	3709	(0.139s)	3709	(49.827s)
credit_tracker	256	(0.028s)	263	(0.287s)
routing_logic_dim_order	191	(0.009s)	191	(0.033s)
wf_alloc_core_rep	205637	(56.194s)	—	(—)

TABLE II: Implication analysis results

We have developed a tool that reads the per-assertion fault coverage statistics generated in the previous step and selects sets of assertions using the following greedy algorithm:

1) If the desired fault coverage is reached, terminate.
2) Pick the assertion which covers the most faults.
3) Add the faults it covers to the "covered faults" set.
4) Remove its faults from all remaining assertions.
5) Go to step 1.

By selecting the assertion with the maximum incremental coverage gain at each iteration, this algorithm generates a minimum-cost subset of assertions that yields each achievable level of coverage.

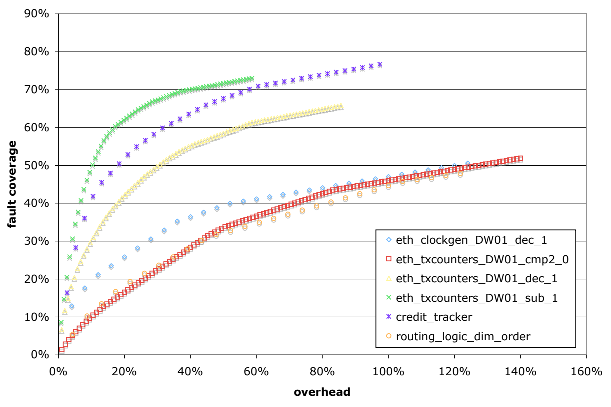


Fig. 2: Fault coverage vs. hardware overhead

Circuit	Max. Coverage	(Overhead)
eth_clockgen_DW01_dec_1	50.6%	(124.0%)
eth_txcounters_DW01_cmp2_0	51.9%	(140.0%)
eth_txcounters_DW01_dec_1	65.8%	(85.6%)
eth_txcounters_DW01_sub_1	73.0%	(58.6%)
credit_tracker	76.6%	(97.4%)
routing_logic_dim_order	47.9%	(121.7%)
wf_alloc_core_rep	55.4%	(—)

TABLE III: Fault coverage limits

IV. Results and Analysis

To evaluate our approach, we applied it to a set of circuits that were extracted from an Ethernet controller and a Network-on-Chip router. Table I presents a brief overview of the various circuits considered and their key characteristics. We used Synopsys Design Compiler to to generate netlists from the circuits' RTL descriptions. In order to limit the complexity of our circuit analysis code, we constrained synthesis to avoid the use of complex logic gates such as AOIs and OAIs.

The results of performing implication analysis on each circuit are presented in table II. The table shows the total number of found implications as well as the CPU time required for analysis for each circuit. Results are given for two separate runs: one with a recursion limit of 8 states, and one with a recursion limit of 1024 states[2]. As can be seen from the results, the vast majority of implications are successfully detected even when running with the tighter recursion limit, at a fraction of the CPU time required for the 1024-state run.

The comparator circuit exhibits an interesting property: it actually produces slightly more assertions for the more limited run; i.e., some implications found in the former are not found in the more extensive run. In such a case, some of the implications detected in the limited run cannot actually occur in the circuit due to a logical contradiction. The more limited run does not explore the state space far enough to detect this contradiction, whereas it is detected in the more extensive run, and as a result, the assertions are removed[3].

Figure 2 shows the coverage achieved for each circuit as a function of the hardware overhead incurred by the addition of assertion checker modules[4]; an overview of the maximum fault coverage that can be achieved using the proposed implication technique is presented in table III.

Two groups can be identified among the results for the various circuits: the smaller decrementer, the comparator and the routing logic are characterized by flat curves, low maximum coverage and a high hardware overhead required to reach that coverage; on the other hand, the larger of the two decrementers, the subtractor and the credit tracking state machine achieve significantly better coverage with a much more reasonable associated cost penalty. Cross-referencing the results from table III with the data in table I suggests that, overall, circuits with lower I/O-to-gate or I/O-to-net ratios tend to achieve better fault coverage and lower hardware overhead compared to circuits with a relatively higher share of inputs and outputs.

Due to the large amount of implications found in the wf_alloc_core_rep circuit, it proved infeasible to exhaustively perform fault simulation for all of them. Consequently, the fault coverage given in table III represents an upper bound; it was generated by performing fault simulation on a netlist that has checkers for all identified assertions included. Figure 2 does not show data points for this particular circuit at all.

This highlights a serious scalability issue of our current approach: complex circuits tend to provide vast amounts of assertions, and running TetraMax NG for all associated hardware checkers becomes prohibitively expensive very fast. Consequently, we must pick a suitable subset of assertions to continue our analysis in such cases. Since we cannot really make informed decisions about which assertions to pick

[2]For the second case, no results are available for the wavefront allocator, as the run exceeded the memory capacity of our test machine.

[3]It should be noted that the presence of these assertions causes no harm; they do, however, increase cost without providing any benefit.

[4]For simplicity, we use the number of gates required for the checkers divided by the number of gates in the original netlist as our overhead metric.

Circuit	Description	# Gates	# Nets	# I/Os
eth_clockgen_DW01_dec_1	8-bit decrementer	25	35	18
eth_txcounters_DW01_cmp2_0	32-bit comparator	95	164	70
eth_txcounters_DW01_dec_1	19-bit decrementer	104	126	40
eth_txcounters_DW01_sub_1	17-bit subtractor	116	154	55
credit_tracker	credit tracking state machine	38	46	13
routing_logic_dim_order	address decoding logic	23	34	26
wf_alloc_core_rep	wavefront allocator	5225	5338	223

TABLE I: Example circuits

without performing fault simulation in the first place, we need to either pick assertions at random, or develop a heuristic that would allow us to identify promising assertions to keep for further analysis; further research is required to investigate these alternatives.

Overall, our results indicate that the fault coverage that can be achieved using gate-level assertions is highly dependent on the properties of the circuit in question, and is in most cases not sufficient to be able to completely displace conventional circuit testing methods. Instead, the assertions fulfill a more complementary role: Assertions would be used for those faults that can be covered with reasonable hardware overhead, and traditional testing methods would be used for the remaining ones. Besides providing limited BIST capabilities, this allows for a significant reduction in test data volume, as no output response signatures have to be stored and compared for those faults that can be covered using assertions.

V. Conclusions and Future Work

In this work, we have explored the use of gate-level assertions for error checking in logic circuits. We have investigated issues of fault coverage and implementation cost. The degree of fault coverage that can be achieved using hardware-based assertion checkers is highly dependent on the characteristics of the circuit, particularly the ratio of inputs and outputs to gates or nets in the design. For many circuits, the level of fault coverage that can be achieved is not high enough to make this approach suitable as a complete replacement for traditional testing techniques. However, gate-level assertion checkers look promising as a complementary technique to the latter ones, providing limited BIST capabilities and allowing for a reduction in test data volume and potentially scan chain complexity.

The discussion in our work relates to first-order assertions; i.e., each implication is from one net-value assignment to another one. One opportunity for future work would be to extend our approach to handle higher-order assertions, in which a combination of multiple net-value assignments implies another one. Such assertions can represent more complex relationships between nets, and should thus enable higher coverage; however, this comes at the cost of significantly increased computational complexity for assertion extraction and higher hardware overhead for the corresponding checkers.

Another promising area for future work relates to algorithmic improvements. As an example, a heuristic-based approach could be used to guide the state space exploration process during logic value backpropagation towards parts of the network that—based on the network structure—are more likely to yield valuable implications. Furthermore, the greedy assertion selection algorithm described in section III could be replaced by a more efficient approach based e.g. on dynamic programming.

Finally, in order to be able to handle complex circuits which have too many implications for exhaustive exploration to be feasible, a heuristic approach is needed for determining a suitable subset of promising implications prior to fault simulation.

Acknowledgment

The authors would like to thank Prof. Mitra for providing us with the opportunity to explore this interesting topic in the course of our EE386 class project. We would also like to thank Sung-Boem Park for his helpful comments and suggestions.

References

[1] M. H. Schulz, E. Trischler, and T. M. Sarfert, "Socrates: A highly efficient automatic test pattern generation system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 1, pp. 126–137, 1988. [Online]. Available: http://ieeexplore.ieee.org/iel1/43/177/00003140.pdf?tp=&isnumber=177&arnumber=3140&punumber=43

[2] W. Kunz and D. K. Pradhan, "Recursive learning: A new implication technique for efficient solutions to cad problems - test, verification, and optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 9, pp. 1143–1158, 1994. [Online]. Available: http://ieeexplore.ieee.org/iel1/43/7534/00310903.pdf?tp=&isnumber=7534&arnumber=310903&punumber=43

[3] S. A. Seshia, W. Li, and S. Mitra, "Verification-guided soft error resilience," *DATE '07: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition 2007*, pp. 1442–1447, 2007. [Online]. Available: http://ieeexplore.ieee.org/iel5/4211748/4211749/04212011.pdf?tp=&isnumber=4211749&arnumber=4212011&punumber=4211748

[4] M. Boulé, J.-S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," *ISQED '07: Proceedings of the 8th International Symposium on Quality Electronic Design*, pp. 613–620, 2007. [Online]. Available: http://ieeexplore.ieee.org/iel5/4148982/4148983/04149103.pdf?tp=&isnumber=4148983&arnumber=4149103&punumber=4148982

[5] M. Boulé and Z. Zilic, "Incorporating efficient assertion checkers into hardware emulation," *ICCD '05: Proceedings of the 2005 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 221–228, 2005. [Online]. Available: http://ieeexplore.ieee.org/iel5/10219/32585/01524156.pdf?tp=&isnumber=&arnumber=1524156

[6] M. Boulé, J.-S. Chenard, and Z. Zilic, "Adding debug enhancements to assertion checkers for hardware emulation and silicon debug," *ICCD '06: Proceedings of the 2006 IEEE International Conference on Computer Design*, pp. 294–299, 2006. [Online]. Available: http://ieeexplore.ieee.org/iel5/4380776/4380777/04380831.pdf?tp=&isnumber=4380777&arnumber=4380831&punumber=4380776

[7] M. Boulé and Z. Zilic, "Efficient automata-based assertion-checker synthesis of psl properties," *Proceedings of the Eleventh Annual IEEE International High-Level Design Validation and Test Workshop*, pp. 69–76, 2006. [Online]. Available: http://ieeexplore.ieee.org/iel5/4110044/4110045/04110065.pdf?tp=&isnumber=4110045&arnumber=4110065&punumber=4110044