Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug

Marc Boulé, Jean-Samuel Chenard and Zeljko Zilic
McGill University
marc.boule@elf.mcgill.ca, {jsamch,zeljko}@macs.ece.mcgill.ca

***Abstract*— This paper presents techniques that enhance automatically generated hardware assertion checkers to facilitate debugging within the assertion-based verification paradigm. Starting with techniques based on dependency graphs, we construct the algorithms for counting and monitoring the activity of checkers, monitoring assertion completion, as well as introduce the concept of assertion threading. These debugging enhancements offer increased traceability and observability within assertion checkers, as well as the improved metrics relating to the coverage of assertion checkers. The proposed techniques have been successfully incorporated into the MBAC checker generator.**

I. Introduction

Assertion-Based Verification (ABV) has recently gained significant popularity in tackling an enormous task of verifying and taping out a successful design. ABV leverages the expressive power of temporal languages to insert checkers that monitor any deviation from the specification's intent. As design size increases, so do the requirements for higher simulation speed and faster regressions. Hardware emulation offers a significant increase in speed, but often at the expense of a more limited observability of the circuit nodes. A methodology based on assertions in the source code must consider the potential use of hardware emulation, therefore the assertions should be amenable to hardware acceleration. Furthermore, when design errors are found during emulation, additional information from the assertion checkers should be used to point to the source of the problem.

This paper enhances previous work on hardware assertion-checker generation [1] to provide further debugging utility of the assertion checkers. The primary context of this work is in the pre-tapeout verification by emulation, however post-layout silicon debug tasks can equally benefit from the proposed methods. Assertion checkers (also called *assertion circuits*) are circuits responsible for monitoring specific properties that a design should respect. Assertions about the design properties are specified at a higher level of abstraction using assertion languages. Two of the most widely used modern assertion languages are the Property Specification Language (PSL) and the SystemVerilog Assertions (SVA).

This work presents the techniques by which assertion checkers synthesized from PSL are enhanced with several key debug features: hardware coverage monitors, activity tracers, assertion completion and assertion threading. The added circuitry can significantly improve the debugging capabilities of the resulting checkers at the expense of a slight increase in the assertion-circuit size.

II. Background: Assertion Checker Generation

Assertions are statements that specify how a given circuit should behave under a variety of circumstances. A *checker generator* ([1], [2]) is a tool which accepts assertion statements and generates monitor circuits that can be used for in-circuit verification. The debugging enhancements introduced in this paper are implemented in our checker generator called MBAC. Assertion checkers are a must for combining hardware emulation or silicon debug with assertion-based verification, given that assertions are not written in Hardware Description Languages (HDLs), but rather in languages such as PSL. Since the assertion languages are powerful and do not necessarily lend themselves to simple checker circuits, the checker generator's task is therefore to transform assertions into efficient monitor circuits that detect assertion failures. Coding checkers by hand can be a tedious and error-prone task. In certain cases, a single PSL statement can imply tens or even hundreds of lines of RTL code in the corresponding checker.

A brief overview of assertion-language features is given next for the Property Specification Language [3], although the key ideas presented in this paper apply equally to SVA or other modern assertion languages. PSL is a powerful language built on several layers of description. The *Boolean layer* consists of the Boolean expressions of the underlying HDL (we use the Verilog "flavor" in this paper). The temporal layer defines *temporal sequences*, with its own set of operators for constructing complex temporal chains of Boolean expressions. Such statements rely on the clock signal to advance time; PSL syntax uses the semicolon to denote a single clock cycle step. The most general meaning of ; is a concatenation of sequences in time. Repetition of events can be bounded ($[*low{:}high]$) or unbounded ($[*]$, which means $[*0{:}\infty]$), and can be applied to Boolean expressions or other sequences. As in regular expressions, $[+]$ denotes a repetition of one or more instances.
Example 1: (PSL Sequences.) If b_i are Boolean expressions, the following are valid PSL sequences:

- $\{b_1; b_2; b_3\}$
- $\{\{b_1; b_2[*]\}\ \&\&\ \{b_3; b_4\}\}$
- $\{\{b_1; b_2\} : \{b_3\}\}$
- $\{\{b_1[*1{:}5]; b_2\}\ \mid\ \{b_3 : b_4[+]\}\}$

The first sequence expresses the fact that the expression b_1 must be true first, followed by b_2 asserted in the next cycle, and by b_3 asserted in the third cycle. Sequence disjunction ($|$) and sequence intersection ($\&$) correspond to the straightforward OR-ing and AND-ing of independent sequences, while length-matching intersection ($\&\&$) requires that the two sequences also be of the same duration. The *fusion* operator (:) is a

concatenation for which the last Boolean primitive of the left side must intersect (i.e. both must be true) with the first primitive of the right side argument.

The temporal layer also includes *properties* which add more expressive power to the use of sequences and Boolean expressions. First, sequences and Boolean expressions are valid properties. The operators never and always indicate properties that either never happen, or happen every clock cycle. The eventually! operator creates a liveness property which can only fail at the end of execution if its sequence argument has not occurred.

Example 2: (PSL Properties.) If $bool$ is a Boolean expression, seq is a sequence and $prop$ is a property, the following are PSL properties:

- never seq
- always $bool$ –> $prop$
- eventually! seq
- always seq |–> $prop$

The |–> operator is known as suffix implication: upon detecting the sequence on the left-hand side, the right-hand side has to hold. A violation in the property will cause the assertion to fail. The |=> is a non-overlapped implication, meaning that the property has to hold on the clock cycle that follows the sequence's occurrence.

PSL's *verification layer* includes directives which instruct a verification tool on how to utilize properties and sequences. The two main operators are:

- assert $prop$;
- cover seq;

The result of using the assert operator is a single bit signal which indicates pass or fail. In dynamic verification, this signal is normally deasserted, and it triggers each time a violation is observed. The cover statement also generates a signal, which in this case will only trigger at the end of execution if its sequence argument never occurred (coverage failure).

The role of the checker generator is to generate an RTL circuit which implements the behavior of an assertion. The latest version of our checker generator builds various automata for properties and sequences. The automata-based checkers can be seen as pattern matching machines, with some similarities to classical regular expression matching. A short description of automata for assertions is given next.

An automaton is often depicted by a directed graph, where vertices are states, and the conditions for transitions among the states are inscribed on edges [4]. In our case, the transition conditions will be expressed by possibly-complex Boolean-layer expressions. For a given assignment of various symbols, all conditions that are true will cause a transition into a *set* of states, leading to non-determinism. A property can be converted to an equivalent finite automaton in a recursive manner [5]. First, terminal automata are built for the Boolean expressions. Next, these automata are recursively combined according to the operators used in a given property. This produces a Nondeterministic Finite Automaton (NFA).

When the automaton representing an assertion reaches a final state, an assertion violation has been found. The checkers implemented in hardware can be derived from the NFA description, provided that implementations allow multiple states to be simultaneously active. Alternatively, deterministic finite automata (DFA) can be produced for purely deterministic sequential implementations; however DFAs are usually larger than equivalent NFAs. We note that because of the way our automata algorithms are designed, the assertion result is not simply a yes/no answer given at the end of execution, but rather a continuous and dynamic report of when/where the assertion has failed, which is obviously more useful for debugging purposes.

III. Debugging in Emulation Environments

Modern interfaces built around IP cores often use pipelined data transfers. Thus, a failure is often embedded in multiple previous transactions running concurrently. Recent work has shown that specifying the various types of transfers, phases, corner scenarios and transfer sequences in a hierarchical fashion can be used to automate the generation of protocol monitors [6] used to assist in tracing back to the source of the failure. We solve this problem using a novel approach explained in Section IV-E. Finally, assistance can be given to the designer by using advanced RTL source code localization techniques such as those recently discussed by Peischl and Wotawa [7].

Tools from companies such as Novas [8] now support advanced debugging methods to help find the root cause(s) of failures by back-tracing assertion failures in the RTL code. Design slicing can further help by automatically reducing the search space by elimination of the circuit elements that are not related to the assertion [9].

For debugging purposes, it may be beneficial to re-create a part of the execution that failed on the emulation environment in a software simulator. Since the assertion checkers from MBAC are generated as RTL code, they can be used both in simulation and in emulation. Leveraging both the observability of the software simulation and the rapid execution of emulated hardware can be done through the use of cut-based debugging [10].

A. A Case for Assertion Space Debugging

In this paper, we primarily consider adding debugging capabilities in the *assertion space*, in which the assertion's faulty behavior is further explored to locate the source of the problem. Our approach is built on providing software control during the checker generation process, which lets the tool instrument the checkers with enhancements that help converge to the root cause of an *assertion failure*. This information could also be used in the circuit space debugging, which we omit from consideration here. The next example shows a simple assertion that might require a fair amount of investigation to deduce the cause of a failure.

Example 3: (Typical Bus Arbitration Assertion.)

assert always { REQ & READY } |–>
{∼GNT ; {BUSY & ∼GNT}[∗0:4] ; GNT & ∼BUSY};

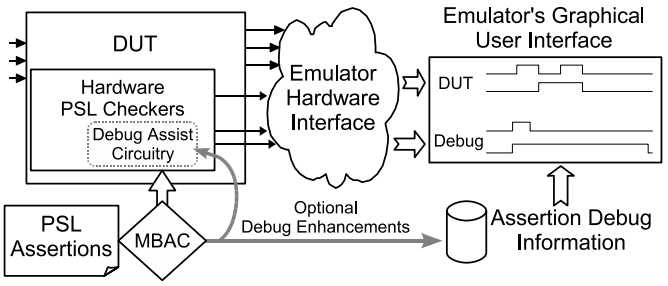

Fig. 1. Hardware PSL Checker with Debugging Enhancements

In this case, the knowledge of the assertion failure will not reveal the exact sequence of events responsible for the failure. For example, if REQ, READY and GNT are all asserted simultaneously, this will be a failure as much as if the GNT was never asserted. Explicit knowledge about the precondition status (in this case "REQ & READY") is better than having to manually create new markers for precondition signals in the debug environment. In our simple example, the precondition marker is easily created; however PSL does not preclude having a complex sequence as a precondition, which would then become a difficult task to re-create in the debug environment.

IV. Enhancing Assertion Checkers for Debug

In this section we present multiple debugging enhancements that can be compiled into the assertion checker circuits produced by our checker generator. These enhancements increase the observability of signals in assertion circuits, and increase the amount of coverage information provided by the checkers. The methodology flow is illustrated in Figure 1. The MBAC checker generator produces assertion-monitoring circuits from PSL statements and augments these checkers with debug-assist circuitry. Other forms of debug information, such as signal dependencies, can also be sent to the front-end applications. Since major FPGA manufacturers now provide hardware interface tools such as embedded logic analyzers, this allows the integration of our techniques even in the simplest, low-end emulation platforms.

A. Dependency Graphs

When debugging an assertion, the ability to quickly determine which signals and parameters can influence the assertion output is an important aid to pinpointing the cause of an error. An explicit enumeration of the signals that can cause an assertion failure helps shorten the debug cycle in any scenario. As a part of the generated checkers in our tool, all of the signal and true parameter dependencies are listed in annotations for each assertion circuit. From this, a dependency graph is then extracted for failure-cause visualization, or for automatic wave script generation in the emulation environment. When an assertion fails, the signals that are referenced in an assertion can be automatically added to the wave window and/or extracted from an emulator, in order to provide the necessary visibility for debugging. Depending on the current capture setup for the trace buffer, an additional rerun might not be needed upon a failure.

Dependency graphs are particularly useful when complex assertions fail, and even more so when an assertion references other user-declared sequences and/or properties, as permitted by the PSL language [3]. In such cases, an assertion's signal dependencies can not be quickly identified at first sight. Optimizations performed on the assertion circuit may result in the removal of some dependencies which will simplify the resulting list. This indicates that either the assertion was incorrectly constructed (since it used redundant signals) or that the combination of multiple sub-expressions resulted in a contradiction or tautology with respect to a subset of the input signals or parameters. In either case, the dependency graph helps in identifying the cause of an assertion violation.

B. Monitoring Activity

Sequences are expressed internally as automata before being converted to the checker hardware. Monitoring the activity of a sequence can be a quick way of knowing whether the input stimulus is actually exercising a portion of an assertion. Using the appropriate compilation option, our tool generates activity signals for each sequence sub-circuit. This activity signal is formed by conjoining all of the state signals in a given sequence automaton, such that when a sequence automaton has at least one active state, the activity signal is asserted. An example of activity signals is visible in Figure 2 for the following assertion.

Example 4: (Test Assertion for Activity Signals.)

$$\text{assert always } (\{a;b\} \mid => \{c[*0{:}1];d\});$$

Here, the activity signals for both sequences are visible, along with the assertion signal (out_mbac), and the assertion as interpreted by Modelsim (gold1). As can be observed, the union of both activity signals coincides with Modelsim's activity indication. Since MBAC's assertion signal is registered, it is asserted on the clock cycle following Modelsim's failure marker (downward triangle). Monitoring activity signals eases debugging by improving observability in assertion circuits. For example, if no activity was ever detected on the right side of a temporal implication, this indicates that the implication is vacuously true [11]; the pre-condition never occurred and thus never triggered the consequent (right side). Monitoring activity for Boolean terminals can also be performed, as required in the property a –> next b, for example.

C. Monitoring Assertion Completion

In order to gauge the effectiveness of a testbench, assertions must be exercised reasonably often in order to be meaningful. After-all, assertions that do not trigger because they were not exercised are not very useful for verification. Conversely, assertions that are extensively exercised but never trigger offer more assurance that the design is operating properly.

Assertions can alternatively be compiled in *completion mode*, whereby the result signal indicates when the assertion completed successfully. The completion mode has no effect

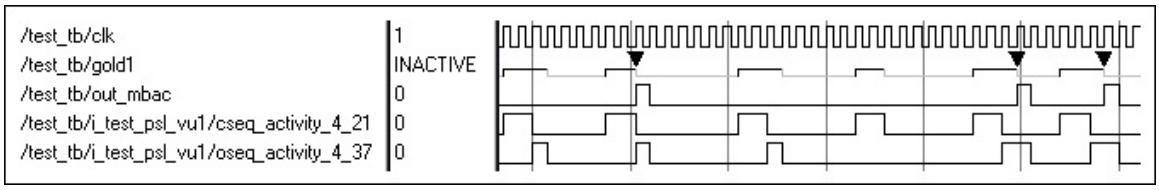

Fig. 2. Activity signals for property: always $(\{a;b\} \mid => \{c[*0{:}1];d\})$. oseq corresponds to the right-side sequence, cseq to the left-side sequence.

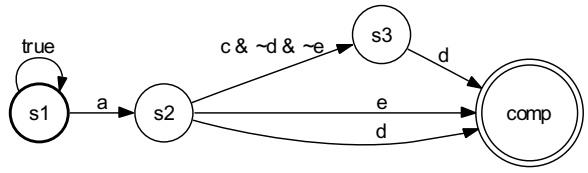

Fig. 3. Debug automaton for always $(\{a\} \mid => \{\{c[*0{:}1];d\}|\{e\})$. The final state indicates when the property completes.

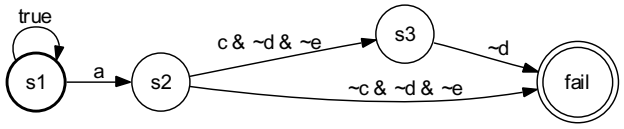

Fig. 4. Automaton for always $(\{a\} \mid => \{\{c[*0{:}1];d\}|\{e\})$. The final state indicates when the property fails.

on assertions of the type assert never seq, given that no obligations are placed on any Boolean expressions. Assertion completion can be visualized using an example.

Example 5: (Test Assertion for Assertion Completion.)

$$\text{assert always } (\{a\} \mid => \{\{c[*0{:}1];d\}|\{e\}\});$$

In completion mode, the consequent automaton (right side of $\mid =>$) is modified to detect the completion of the sequence. For a given start condition, only the first completion is identified by the automaton. An example of completion monitoring is shown in Figure 3. As a reference point, the example assertion is normally compiled as the automaton shown in Figure 4. In this case, the result signal (or final state) is triggered when the assertion fails. The highlighted state s1 indicates the initial state, which is the only active state upon reset. The PSL abort operator has the effect of resetting a portion of the checker circuitry, and thus applies equally to normal mode or completion mode.

D. Counting Activity

MBAC includes options to automatically create counters on *assert* and *cover* statements for counting activity. Counting assertion failures is straightforward, however counting the cover directive requires particular modifications. In dynamic verification, the cover operator is usually rewritten as follows:

$$\text{cover } seq \quad \rightarrow \quad \text{assert eventually! } seq$$

This approach is not compatible with counters (and in general is not useful for debugging) because eventually! is a liveness property and thus triggers only at the end of execution. In order to count occurrences for coverage metrics, a plain detection (or matching) automaton is instead built for the sequence, and a counter is used to count the number of times the sequence occurs. The cover signal then triggers only at the end-of-execution if the counter is at zero. If no counters are desired, a one-bit counter is implicitly used. The counters are built to saturate at their final count and do not roll-over. The counters are also initialized by a reset of the assertion checker circuit, typically the reset of the Device Under Verification (DUV).

Counters can be used with completion mode from Subsection IV-C to construct more detailed coverage metrics for a given testbench. Knowing how many times an assertion completed successfully can be just as useful as knowing how many times an assertion failed. For example, if a predetermined number of a certain type of bus transaction is initiated by a testbench, and an assertion that is responsible for catching a *faulty* sequence never fails, we may deduce that this particular transaction type is working properly. For added sanity checking, the assertion could be compiled in completion mode with a counter, and at the end of the testbench, this count value should correspond to the number of bus transactions that were exercised. In the example, the assertion may have passed because a different kind of transfer was erroneously issued. Completion mode provides a confirmation that if an assertion never failed, it was not because of a lack of stimulus. Counters and completion of assertions are implemented in simulators such as Modelsim, it is therefore natural that these features be also incorporated into assertion circuits.

E. Hardware Assertion Threading

When users want to more closely observe which start condition caused a failure, our checker generator has the ability to instantiate many copies of sequence circuits, and alternately dispatch preconditions into succeeding circuits. This allows a violation condition to be extracted from the other pipelined signals in the assertion circuit. The assertion threading tries to separate the parallel activity to help identify the root cause of the sequence of events leading to the failure of an assertion. Threading applies to PSL *sequences*, which are the typical means for specifying complex temporal chains of events.

An example scenario where assertion threading is useful is in the verification of highly pipelined circuits, where temporally complex sequences are used in assertions. In such cases, it is highly desirable to partition sequences into different threads in order to separate a failure sequence from other sequences. Assertion threading achieves the effect of creating multiple deterministic state machines, which are more natural to hardware designers and intuitive for debugging.

Figure 5 illustrates the mechanisms used to implement assertion threading. The hardware dispatcher redirects the precondition signal to the multiple sequence-checker units in

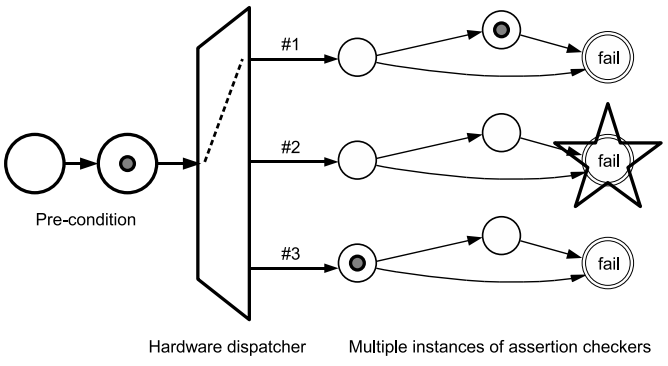

Fig. 5. Hardware Assertion Threading

a round robin sequence. The tokens indicate the progress through the sequence automata. In the example, hardware thread #2 has identified a failure. The knowledge of the current state of the hardware dispatcher (pointing to thread #1), indicates that the assertion was triggered two valid preconditions before the failure. Therefore, the assertion failure can be pinpointed to that particular precondition or any modulo–3 preconditions before. The precondition typically corresponds to the result of the left-side of a temporal implication, whereas the right side is the consequent. Some tradeoff is required between an accurate location of the source of the failure and hardware resources, as will be shown in Section V.

In assertion threading, entire failure-detection sequence-automata are replicated. Since a single automaton will detect all sequence failures, replicating the automaton and sending tokens into different copies ensures that no failure will be missed. The dispatcher uses a logical rotation of a one-hot encoded register such that at each clock cycle a precondition is guaranteed to be passed to one of the hardware threads. If a token enters a thread for which a previous token is still being processed, identifying the precise cause of a failure becomes more difficult; however, no failures will be missed. In such cases, all that must be done is to increase the number of hardware threads in order to properly isolate a sequence. To complete the threading, the sequence output is a disjunction of the threaded automata outputs. Threading also applies to sequences in the left side of a temporal implication. In such cases, normal detection sequence automata (as opposed to failure detection) are replicated. Seen from the sub-circuit boundary, a multi-threaded sub-circuit's behavior is identical to that of a non-threaded sub-circuit.

V. Experimental results

The effects of assertion threading, assertion completion and activity monitors are explored by synthesizing the assertion circuits produced by our checker generator using ISE 6.2.03i from Xilinx, for a XC2V1500–6 FPGA. The dependency graphs from Section IV-A do not influence the circuits generated by the checker generator, while the assertion and coverage counters from Section IV-D contribute a hardware overhead that is easily determined *a priori*. The number of flip-flops (FF) and four-input lookup tables (LUT) required by a circuit is of primary interest, given that assertion circuits are targeted towards hardware emulation and silicon debug. Since speed may also be an issue, the maximum operating frequency for the worst clk-to-clk path is reported. The assertions used in this section were created during the development of MBAC to exercise the checker generator as thoroughly as possible. Typical assertions, such as those used for verifying bus protocols, span few clock cycles and do not showcase the strength of our checker generator because they are easily handled.

A. Assertion Completion and Activity Monitoring

The activity monitors introduced in Section IV-B are used to observe when sequences are undertaking a matching or a failure detection. An activity signal is composed of the disjunction of state signals from all of the states in a given automaton. Table I shows the resource usage of example assertions with and without the addition of sequence-activity monitors. As can be noticed, the maximum operating frequency is virtually not affected, and in some cases, an additional flip-flop is required. The effect of the OR gate required for the state-signal disjunction is visible in the LUT metric. Further benchmarking shows the efficiency of our checker generator, compared to the FoCs checker generator from IBM [2], [12].

As described in Section IV-C, assertions can also be compiled in completion mode as opposed to the typical failure mode. Table II shows hardware metrics for a set of example assertions compiled in normal mode and in completion mode. From the table, it can be observed that a completion-mode assertion utilizes an equal or smaller amount of resources.

B. Assertion Threading

As explained in Section IV-E, assertion threading replicates sequence circuits in order for the failure conditions to be isolated from other preconditions. This was shown to ease the debugging process considerably, particularly when temporally complex assertions are used. Table III shows how the resource utilization scales as a function of the number of hardware threads. Because 8-way threading is only useful for sequences that span at least 8 clock cycles, the assertions used must have a certain amount of temporal complexity for the results to be meaningful. From the table, it can be observed that resource utilization scales linearly with the number of hardware threads.

VI. Conclusion

In this paper we have presented techniques that facilitate debugging within the ABV framework, either in the emulation or in the silicon debug stages. By selecting various compilation options, debugging is enhanced by providing better observability, traceability and coverage metrics in the assertion checkers generated by MBAC. While providing an increased ability to determine the causes of errors, the hardware overhead is modest. These improvements are particularly well suited for the complex temporal sequences of modern assertion languages.

TABLE I

RESOURCE USAGE OF ASSERTION CIRCUITS AND ACTIVITY MONITORS. (N.O. = NO OUTPUT)

Assertion	FoCs			MBAC			MBAC+Act.Mon.		
	FF	LUT	MHz	FF	LUT	MHz	FF	LUT	MHz
Bus arbitration assertion from Example 3	6	13	428	6	**10**	**487**	7	11	487
assert always ({a;b} |=> {c[*0:1];d}); (Example 4)	4	4	622	4	4	622	4	6	616
assert always ({a} |=> {{c[*0:1];d}|{e}}); (Example 5)	3	4	622	3	4	622	4	5	622
assert never {a;d;{b;a}[*2:4];c;d};	25	24	622	**12**	**12**	622	12	16	616
assert always {a} |=> {e;d;{b;e}[*2:4];c;d};	N.O.			**15**	**21**	**378**	16	25	375
assert always {a} |=> {b; {c[*0:2]} | {d[*0:2]} ; e};	7	12	355	7	12	**428**	8	14	425
assert never { {{b;c[*1:2];d}[+]} && {b;{e[–>2:3]};d} };	43	51	422	**16**	**20**	422	16	25	419
assert always {a} |=> {{{c[*1:2];d}[+]} && {{e[–>2:3]};d}};	N.O.			**16**	**39**	**349**	18	44	337
assert always {a} |=> {{{b;c[*1:2];d}[+]} & {b;{e[–>2:3]};d}};	N.O.			**44**	**127**	**269**	45	136	268
assert always {a} |=> {{{b;c[*1:2];d}[+]} && {b;{e[–>2:3]};d}};	N.O.			**35**	**112**	**258**	36	119	270

TABLE II

ASSERTION-CIRCUIT RESOURCE USAGE IN TWO MBAC MODES.

Assertion	Normal			Assertion Completion		
	FF	LUT	MHz	FF	LUT	MHz
Bus Arbitration Assertion from Example 3	6	10	487	6	**8**	487
assert always ({a;b} |=> {c[*0:1];d}); (Example 4)	4	4	622	4	4	622
assert always ({a} |=> {{c[*0:1];d}|{e}}); (Example 5)	3	4	622	3	**3**	622
assert always {a} |=> {e;d;{b;e}[*2:4];c;d};	15	21	378	15	**16**	**483**
assert always {a} |=> {b; {c[*0:2]} | {d[*0:2]} ; e};	7	12	**428**	7	**10**	425
assert always {{{b;c[*1:2];d}[+]} : {b;{e[–>]};d}} |=> next a;	8	8	487	8	8	487
assert always {a} |=> {{{c[*1:2];d}[+]} && {{e[–>2:3]};d}};	16	39	**349**	16	**31**	339
assert always {a} |=> {{{b;c[*1:2];d}[+]} & {b;{e[–>2:3]};d}};	44	127	269	44	127	269
assert always {a} |=> {{{b;c[*1:2];d}[+]} && {b;{e[–>2:3]};d}};	35	112	258	35	**96**	**299**

TABLE III

AREA TRADEOFF METRICS FOR ASSERTION THREADING.

Assertion	None			2-way			4-way			8-way		
	FF	LUT	MHz	FF	LUT	MHz	FF	LUT	MHz	FF	LUT	MHz
Example 3	**6**	**10**	**487**	15	19	378	29	38	300	57	69	258
A1	**12**	**12**	622	25	24	622	49	47	521	97	93	448
A2	**15**	**21**	**378**	33	46	326	65	85	273	129	176	236
A3	**16**	**20**	422	33	40	422	65	79	422	129	156	355
A4	**26**	**76**	**278**	57	150	267	113	290	233	225	579	223
A5	**35**	**112**	**258**	73	224	242	145	445	227	289	856	198
A1: assert never {a;d;{b;a}[*2:4];c;d};												
A2: assert always {a} |=> {e;d;{b;e}[*2:4];c;d};												
A3: assert never { {{b;c[*1:2];d}[+]} && {b;{e[–>2:3]};d} };												
A4: assert always {a} |=> {{{b;c[*1:2];d}[+]} : {b;{e[–>]};d}};												
A5: assert always {a} |=> { {{b;c[*1:2];d}[+]} && {b;{e[–>2:3]};d} };												

REFERENCES

[1] M. Boule and Z. Zilic, "Incorporating efficient assertion checkers into hardware emulation," in *IEEE International Conference on Computer Design*, Oct. 2005, pp. 221–228.

[2] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, "FoCs: Automatic Generation of Simulation Checkers from Formal Specifications," *Conference on Computer Aided Verification*, pp. 538–542, 2000.

[3] Accellera. (2005) PSL Language Reference Manual, version 1.1. [Online]. Available: http://www.eda.org/vfv/docs/PSL-v1.1.pdf

[4] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages and Computation, 2nd ed.* Addison-Wesley, 2000.

[5] M. Boule and Z. Zilic, "Efficient automata-based assertion-checker synthesis of PSL properties," *Under review*, 2006.

[6] A. Nandi, B. Pal, N. Chhetan, P. Dasgupta, and P. P. Chakrabarti, "H-DBUG: A high-level debugging framework for protocol verification using assertions," in *IEEE Indicon Conference*, Chennai, India, Dec. 2005, pp. 115–118.

[7] B. Peischl and F. Wotawa, "Automated source-level error localization in hardware designs," in *IEEE Design and Test of Computers*, vol. 23, no. 1, Jan. 2006, pp. 8–19.

[8] Y.-C. Hsu, B. Tabbara, Y.-A. Chen, and F. Tsai, "Advanced techniques for RTL debugging," in *Design Automation Conf.*, 2003, pp. 362–367.

[9] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum, "Program slicing of hardware description languages," in *Conference on Correct Hardware Design and Verification Methods*, 1999, pp. 298–312.

[10] D. Kirovski, M. Potkonjak, and L. M. Guerra, "Improving the observability and controllability of datapaths for emulation-based debugging," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 11, Nov. 1999, pp. 1529–1541.

[11] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient Detection of Vacuity in Temporal Model Checking," *Formal Methods in System Design*, pp. 141–163, 2001.

[12] IBM AlphaWorks. (2006) FoCs Property Checkers Generator ver. 2.03. [Online]. Available: http://www.alphaworks.ibm.com/tech/FoCs