SILICON DEBUG AND DIAGNOSIS

Debug enhancements in assertion-checker generation

M. Boulé, J.-S. Chenard and Z. Zilic

Abstract: Although assertions are a great tool for aiding debugging in the design and implementation verification stages, their use in silicon debug has been limited so far. A set of techniques for debugging with the assertions in either pre-silicon or post-silicon scenarios are discussed. Presented are features such as assertion threading, activity monitors, assertion and cover counters and completion mode assertions. The common goal of these checker enhancements is to provide better and more diversified ways to achieve visibility within the assertion circuits, which, in turn, lead to more efficient circuit debugging. Experimental results show that such modifications can be done with modest checker hardware overhead.

1 Introduction

Techniques for post-fabrication debugging, known as silicon debugging, are receiving much attention, as increasing transistor counts and smaller process technologies make it difficult to achieve correct silicon. Companies such as DAFCA, for example, allow the addition of register transfer level (RTL) silicon-debug instrumentation to the source design, the status of which can be read back through the Joint Test Action Group (JTAG) interface [1]. Examples of useful debug instruments that are implemented in their tools are: in-circuit trace buffers for capturing signals or supplying vectors, signal probe multiplexers and logic analyser circuitry. To ensure flexibility in providing these post-silicon debug instruments, they are implemented in small blocks of added programmable logic gates. The debugging instruments [1] and the checker enhancements presented in this paper encompass a collection of techniques that share a common goal, to help increase the efficiency of the debugging process. The boldface numbers in the tables show the best result for each test case. For FFs and LUTs, lower is better; for speed (MHz), higher is better.

Verification aims at eliminating errors before tape-out, by ensuring that a design follows its intended specification. Assertion-based verification (ABV) is a relatively new methodology that is becoming increasingly important [2]. Assertions are meant to express intended circuit functionality in a formal language. The two most common assertion languages are the property specification language (PSL) and SystemVerilog assertions (SVA). Assertion failures reveal design errors either through formal or simulation-based verification, and are an important failure localisation mechanism aiding the debugging process. Increasingly, formal verification tools and simulators are able to interpret assertions, which help pinpoint design errors before fabrication. Assertions can also be used in hardware emulation or simulation accelerators; however, descriptions in high-level assertion languages are not easily converted into efficient RTL descriptions suitable for emulation platforms, such as FPGA-based emulators. To exploit the power of assertions in hardware form, a checker generator [3] was designed to create efficient circuit-level checkers from assertion statements. These checkers monitor the device under verification (DUV) for violations of assertions and raise an output signal when a violation is observed.

Circuit-level assertion checkers can be used not only for pre-fabrication verification, but also for post-fabrication silicon debugging (Fig. 1). Assertion checkers can be purposely added to the synthesised design to increase debug visibility during initial testing of the IC. Assertions compiled with a checker generator can also be used as on-line circuits for various in-field status monitoring purposes, as shown in the right side of Fig. 1.

In the emerging design for debug (DFD) space, several companies are promoting a range of solutions. Tools from companies such as Novas now support advanced debugging methods to help find the root cause(s) of errors by back-tracing assertion failures in the RTL code [4]. Temento's DiaLite product accepts assertions and provides in-circuit debugging features. DAFCA also offers this possibility, and provides support for assertion checker synthesis and use. However, as these tools are from commercial ventures, papers seldom disclose their actual inner-workings.

Increasing and enhancing the visibility into the design's signals is also an important aspect in silicon debugging and DFD [5]. In this paper, increasing visibility using ABV techniques is explored. More specifically, this work presents the techniques that enhance assertion checkers with several debug features [6]: hardware coverage monitors, activity tracers, assertion completion and assertion threading. These enhancements improve the debugging capabilities of the resulting checkers in all scenarios in shown Fig. 1. In verification and silicon debugging, the enhancements offer the means to help pinpoint the exact cause of an assertion failure. In the on-line monitoring scenario, the assertion completion mode and the activity monitors can play a key role in creating checkers that evaluate the quality of a circuit for in-field, real-time diagnosis. All debugging enhancements are implemented in our checker generator called MBAC, where they can be

© The Institution of Engineering and Technology 2007
doi:10.1049/iet-cdt:20060209

Paper first received 21st November 2006 and in revised form 20th July 2007
The authors are with the McGill University, Montréal, Québec, Canada

E-mail: marc.boule@elf.mcgill.ca

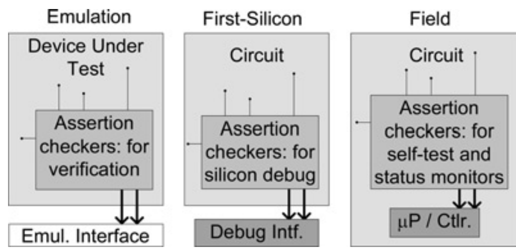

Fig. 1 *Usage scenarios for hardware assertion checkers*

optionally activated. The added visibility into parts of the assertion checker circuits, along with additional ways to track assertion results, are the principal means by which we propose to improve the assertion-based debugging process.

The paper is organised as follows. Background on assertions and our checker generator is presented in Section 2. The debug enhancements for assertion checkers are described in Section 3. The effects of these modifications to the checkers are evaluated in the experimental results in Section 4.

2 Background

We now overview the features of PSL [7, 8] that, to a large extent, also apply to SVA or other modern assertion languages. PSL is a powerful language that consists of several layers. The Boolean layer consists of the expressions of the underlying HDL (we use the Verilog 'flavor' in this paper). The temporal layer defines sequences, which are regular expressions over the Boolean layer formulas. This allows the compact specification of complex temporal chains of Boolean expressions. Such statements rely on the clock signal to advance time, and are placed between curly brackets. PSL syntax uses the semicolon to specify the temporal concatenation of two sub-sequences (clock cycle separator). Sequences may be repeated with a Kleene star operator, either infinitely ([*]) or with bounds ([*l:h]). Temporal sequences may also be fused using the colon operator, which constitutes an overlapped concatenation. Sequence intersection such as length-matching (&&) and non-matching (&), as well as disjunction (|), is also possible. Sequences also comprise additional sugaring operators, which are defined in [7].

The temporal layer also defines properties that add more expressive power to sequences and Boolean expressions. Here, simple temporal operators such as always or never are used to enforce how a property or a sequence should behave, respectively. Overlapped and non-overlapped suffix implication (|−> and |=>, respectively) can also be used to create a temporal implication where the antecedent is a sequence, and the consequent is a property. The eventually! operator creates a strong property that triggers at the end of execution if its sequence argument was not observed. Other properties such as abort, next, until, and so on are also defined in [7]. Sequences and Booleans can also be used directly as properties.

The two main directives in PSL's verification layer are assert and cover. In dynamic verification, the result of the assert operator on a property is a pass/fail signal. This signal is normally deasserted, and triggers each time a violation is observed. The cover statement generates a signal that only triggers at the end of execution if its sequence argument never occurred (coverage failure).

A checker generator [3, 9] is a tool that generates monitor circuits (also called checkers) from assertions. The generated checkers should be as small as possible, in order to utilise the smallest amount of circuitry when the checkers are destined for a hardware implementation. Internally, the MBAC checker generator builds various automata for properties [10] and sequences [11]. Automata for each assertion are then transformed to RTL code [11] in a manner similar to hardware pattern matching [12]. An example of this is shown on the right-hand side of Fig. 2*a*. An implementation of the debug enhancements proposed in this paper builds on the latest version of our checker generator.

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 [13]. In our case, the transition conditions constitute Boolean-layer expressions. At each clock cycle, the automaton transitions into a new set of active states, depending on the symbols and the status of the values on their edges. When a final state activates in an assertion automaton, a violation has occurred. A property can be converted to an equivalent finite automaton in an inductive manner [10] as follows. First, terminal automata are built for the Boolean expressions. Next, these automata are recursively combined according to the operators used in a given property.

The debugging capabilities we introduce are in the assertion domain, in which the assertion's behaviour is further explored to locate the source of the problem. Our approach

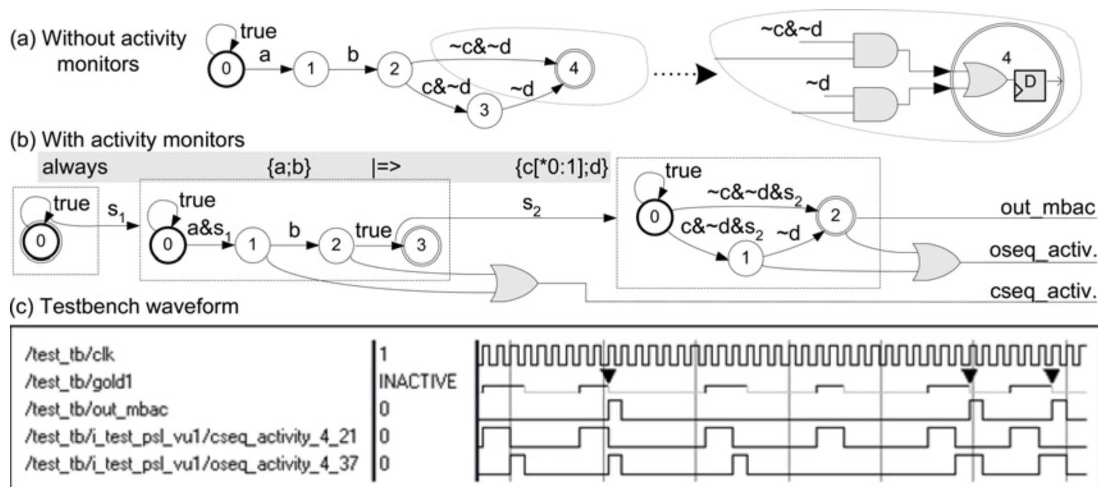

Fig. 2 *Activity signals for property: always ({a;b}|=>{c[*0:1];d}) oseq corresponds to the right-side sequence, cseq to the left-side sequence*

involves augmenting the checker generation algorithms to instrument the checkers in ways that help to locate the root causes of assertion and circuit failures. The following example shows a simple assertion that might require a fair amount of investigation to deduce the cause of a failure.

Example 1: A typical bus arbitration assertion. The assertion below states that whenever the arbiter is ready and receives a bus request, then the grant signal should be low in that cycle (the temporal implication |–> is non-overlapping). The grant should then be given within at most five cycles; furthermore, the arbiter's busy signal must be true until the grant is given, when it must then be low.

$$
\begin{array}{l}
\textsf{assert always}\{\mathrm{REQ}\,\&\,\mathrm{READY}\}|\!-\!>\\
\quad\{\sim\mathrm{GNT};\ \{\mathrm{BUSY}\,\&\ \sim\mathrm{GNT}\}[*0{:}4];\\
\quad\mathrm{GNT}\,\&\,\sim\mathrm{BUSY}\};
\end{array}
$$

Knowing only that this assertion fails will not reveal the exact cause, or even the sequence of events responsible for the assertion 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. If a tool can provide the explicit knowledge about the antecedent's status (in this case 'REQ & READY'), this avoids having to create new signals in the debug environment for monitoring antecedent signals manually. In our simple example, the antecedent signal can be easily created manually; however, PSL allows having a complex sequence as an antecedent, which would then be difficult to re-create in the debug environment.

3 Debug enhancements for assertion checkers

We now present debugging enhancements that can be optionally added to the assertion checkers produced by our checker generator. These enhancements increase the visibility within assertion circuits, and also enhance the coverage information provided by the checkers. Fig. 3 shows how the checker additions intervene in the verification methodology. The MBAC checker generator produces assertion-monitoring circuits from PSL statements augmented with various debug-assist circuitry. Other forms of debug information, such as signal dependencies, can also be sent to the front-end applications. Since our techniques are implemented at the RTL level within the checkers, they can be used in concert with any other circuit debugging tools.

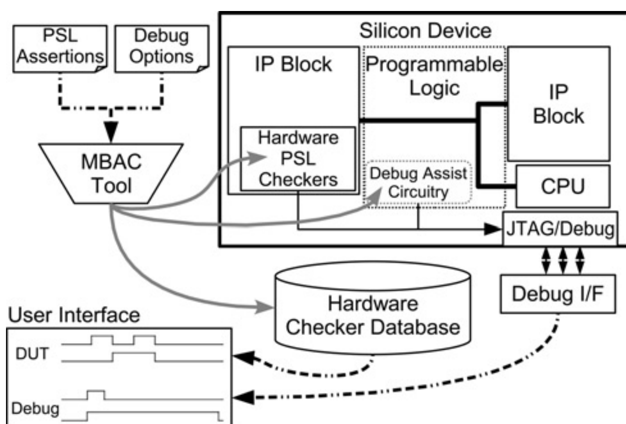

Fig. 3 *Hardware PSL checker with debugging enhancements*

3.1 Dependency graphs

When debugging failed assertions, it is useful to determine quickly which signals and parameters can influence the assertion output. In MBAC, all of the signal and parameter dependencies are listed in annotations for each assertion circuit. A dependency graph is constructed by the tool to help pinpoint the cause of an error, or for automatic wave script generation in an 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. Dependency graphs are particularly useful when complex assertions fail, especially when an assertion references other user-declared sequences and/or properties, as allowed by PSL [7]. In such cases, assertion signal dependencies help point to the source of the problem.

3.2 Signalling assertion completion

For a verification scenario to be meaningful, assertions must be exercised: assertions that do not trigger because the test vectors did not fully exercise them are not very useful for verification or debug. In such cases, that is, when the assertions are trivially true, the designers could be led to believe that the property has been validated, and thus overlook the true cause of a non-failure. On the contrary, assertions that are extensively exercised but never trigger offers more assurance that the design is operating as specified. The dependency graphs from the previous section efficiently determine which signals must be stimulated to exercise properly an assertion that is found to be trivially true.

In MBAC, assertions can be alternatively compiled in a completion mode, to indicate when assertions complete successfully and are not trivially true. The completion mode affects assertions that place obligations on certain sub-expressions, such as the consequent of temporal implications for example. In temporal implications, for each observed antecedent, the consequent must occur or else the assertion will fail. As opposed to indicating the first failure in each consequent, as is usually done, the completion mode assertion indicates the first success in the consequent, for each activation coming from the antecedent.

The completion mode has no effect on assertions such as '`assert never` *seq*', given that no obligations are placed on any Boolean expressions. This assertion states that the sequence argument *seq* should not be matched in any cycle. Thus, every time the sequence is matched (i.e. is detected as occurring), a violation occurs and the assertion output triggers. The actual PSL syntactical elements which are affected by the completion mode are sequences and Boolean expressions when used directly in properties, with the following exceptions: the argument of the `never` operator and the antecedent of implications (suffix implication and property implication).

Using terminology in [14], our technique identifies interesting witnesses, that is, examples of where the property was exercised. Antecedent non-occurrence – commonly referred to as vacuity – is the only one possible cause for trivial validity. The knowledge that an assertion completes successfully can be useful when evaluating the coverage quality of a regression suite. Completion mode creates behaviour analogous to the `cover` operator for sequences, except it is at the property level. Completion mode can also be referred to as 'pass checking', 'success checking' and 'property coverage'.

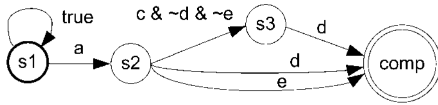

Fig. 4 *Completion automation for always ({a}|=>{{c[*0:1]; d}|{e}})*

We now describe the completion algorithm and illustrate it with an example. The completion mode transformation algorithm first determinises the automaton such that each activation is represented by only one active state. From any given state, a deterministic automaton transitions into at most one successor state. Determinising automata with Boolean expressions on transitions is more involved than in conventional automata [15]. The determinisation step is required so that when the first completion is identified, no other subsequent completions will be reported for the same activation.

The second step in the completion mode algorithm is to remove all outgoing edges of the final states, when applicable (in Fig. 4, there were no such edges to remove). Any unconnected states resulting from this step are removed during automata minimisation. Both the failure and the completion transformation algorithms take as input the detection automaton that corresponds to the sequence being handled. The completion-mode algorithm can also be used to implement the SVA operator first_match(), and is a dual of the FirstFail algorithm in [11]. The FirstFail algorithm is normally applied to Booleans and sequences that are used directly as properties. In completion mode in the checker generator, all calls to the FirstFail algorithm are replaced by calls to the FirstMatch algorithm (also called the completion mode algorithm). Assertion completion is best visualised using an example.

Example 2: Test assertion for assertion completion.

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

The assertion above is normally compiled as the automaton in Fig. 5, where the final state is triggered when the assertion fails. The completion mode automaton for this example is shown in Fig. 4. The sequence of events *a*; *c*; *d*, for example, will make the automaton in Fig. 4 trigger (completion); however, the failure automaton will not reach a final state given that the sequence conforms to the specification indicated by the assertion. In the automata graphs, the highlighted state s1 indicates the initial state, which is the only active state when reset is released. The PSL abort operator has the effect of resetting a portion of the checker circuitry [10], and thus applies equally to normal mode or completion mode.

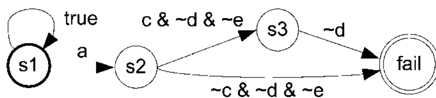

Fig. 5 *Normal automaton for always ({a}|=>{{c[*0:1];d}|{e}})*

3.3 Counting activity

MBAC includes options to create automatically counters on assert and cover statements for counting activity. Counting assertion failures is straightforward, as observed in the top half of Fig. 6; however, counting the cover

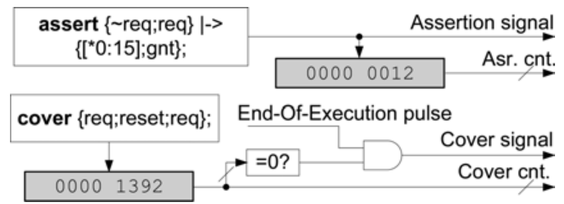

Fig. 6 *Counting assertions and covers*

directive requires some modifications. In dynamic verification, cover is a liveness property which triggers only at the end of execution. In order to count occurrences for coverage metrics, a plain matching (detection) automaton is built for the sequence argument (In PSL, a property is not a valid argument for the cover operator), and a counter is used to count the number of times the sequence is matched. The cover signal only triggers at the end of execution if the counter is at zero, as shown in the lower half of Fig. 6. If no counters are desired, a one-bit counter is implicitly used. The counters are width parameterised, and by threshold arithmetic do not roll-over when the maximal count is reached. The counters are also initialised by a reset of the assertion checker circuit.

Counters can be used with completion mode (Section 3.2) to construct more detailed coverage metrics for a given scenario. 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 pre-determined number of a certain type of bus transaction is initiated, the related assertion should see itself complete successfully the same number of times. In general, by signalling successful witnesses, completion mode provides an indication that if an assertion never failed, it was not because of a lack of proper stimulus.

3.4 Monitoring activity

Sequences are expressed internally as automata before being used to construct an assertion automaton. Monitoring the activity of a sequence is a quick way of knowing whether the input stimulus is actually exercising a portion of an assertion. Activity is defined as a disjunction of all states in an automaton; thus anytime a state is active, the automaton is active. A sequence can show internal activity when undergoing a matching, even if its output does not trigger (i.e. reach an accepting state). Conversely, if a sequence output triggers, the automaton representing it is guaranteed to show internal activity. The dependency graph of an assertion can also be used to pinpoint the root cause of a suspiciously inactive sequence during a verification scenario.

Using the appropriate compilation option, our tool further generates activity signals for each sequence sub-circuit. The only states in sequence automata that are excluded from consideration for activity monitors are: the start state and the final state when a sequence is the antecedent of the |=> operator. The reason for these exceptions is that when a sequence automaton is kept isolated from the rest of the assertion's automaton, its initial state has a self loop with the Boolean true, which does not represent meaningful activity. When a sequence appears as the left side of a non-overlapped suffix implication, it is rewritten to an overlapped implication by concatenating an extra '{true}' sequence element to the end of the sequence [10], which also does not represent meaningful activity. An example

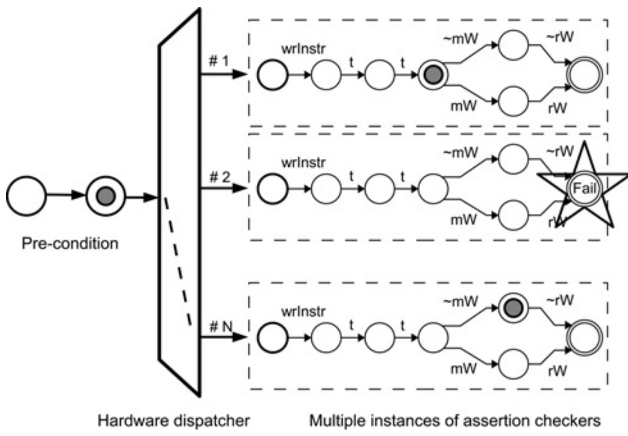

Fig. 7 *Hardware assertion threading*

of activity signals is visible in Fig. 2*c* for the following assertion.

Example 3: Test assertion for activity signals.

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

In Fig. 2*c*, the activity signals for both sequences are visible, along with the assertion signal (out_mbac), and the assertion as interpreted by Modelsim (gold1).

Under normal conditions, each assertion is represented by a single automaton in MBAC, before its transformation to RTL. Fig. 2*a* shows how the example assertion would normally appear as a single automaton when activity monitors are not desired. To implement activity monitors and the assertion threading, discussed in Section 3.5, it is necessary to isolate a sub-automaton so that it is not merged with the remainder of the assertion's automaton during automata optimisations. The automata that are isolated correspond to the sub-expressions that are being monitored and/or threaded, which in our approach correspond to top-level sequences or Boolean expressions appearing in properties.

Monitoring activity signals eases debugging by improving visibility in assertion-circuit processing. For example, an implication whose antecedent is never matched is said to pass vacuously [14]. When monitoring antecedent activity, a permanently inactive antecedent does indicate vacuity, but this seldom occurs given that a single Boolean condition can activate a state within the antecedent. An example to illustrate this is shown in Fig. 2*b*, where state1 in the antecedent automaton is active when *a* is true. For activity monitors to be the most useful, the right side (consequent) needs to instead be monitored because an inactive consequent means that the antecedent was never fully detected (and thus never triggered the consequent). If no activity was ever detected on the consequent of a temporal implication, this indicates that the implication is vacuously true: the antecedent never fully occurred and thus never triggered the consequent. The fact that the antecedent never fully occurred does not mean that there was no activity within it; conversely, activity in the antecedent does not mean that it fully occurred.

3.5 Hardware assertion threading

Assertion threading is a technique by which the checker generator instantiates multiple copies of a sequence checking circuit, and alternately activates these circuits. This allows a violation condition to be separated from the other concurrent activations in the assertion circuit, in order to help visualise which exact start condition caused a failure. In general, by using a single automaton-based recogniser, all temporal checks become intertwined in the automaton during processing. The advantage is that a single automaton can catch all failures; however, the disadvantage is that it becomes more difficult to correlate a given failure with its input conditions. The assertion threading in effect separates the concurrent activity to help identify the root cause of the sequence of events leading to an assertion failure. Threading applies to PSL sequences, which are the typical means for specifying complex temporal chains of events.

Fig. 7 illustrates the mechanisms used to implement assertion threading. The hardware dispatcher redirects the activation signal to the multiple sequence-checker units in a round robin sequence. The tokens indicate the progress through the sequence automata. In the figure, hardware thread #2 has identified a failure. With this information, we can trace back to the antecedent expression that initiated the sequence checking in thread #2. An example will follow illustrating how this method can be used in tracing back an execution error in a CPU.

In assertion threading, entire failure-matching sequence-automata are replicated. Since a single automaton can detect all sequence failures, replicating the automaton and sending tokens into different copies ensures that no failure is missed even if the number of threads is below the concurrency level of the monitored sequence. The dispatcher rotates a one-hot encoded register such that each activation is sent 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. In such cases, increasing the number of hardware threads can help to properly isolate a sequence.

Threading also applies to the plain matching sequence automata (as opposed to the failure matching automaton discussed above). In such cases, the plain occurrence matching automaton is threaded for increased causality visualisation. The nuance between plain matching and failure-matching modes (called conditional and obligation

```
default clock          =  ( posedge Clk );
sequence Smemwr        =  {[*2] ; MemWr; ~RegWr};                              //memory write
sequence Sregwr        =  {[*2] ; ~MemWr; RegWr};                              //register write
sequence Swr_instr     =  {InstrValid && Instruction[31]==1'b1 &&
                           (Instruction[30:29]==2'b10 || Instruction[30:29]==2'b01) };   //write instruction
property Pcorrect_wr   =  always { Swr_instr } |=> { {Smemwr} | {Sregwr} };    //write works
assert Pcorrect_wr;                                                            //assertion
```

Fig. 8 *Assertion for pipelined CPU write instruction (CPU pipeline example)*

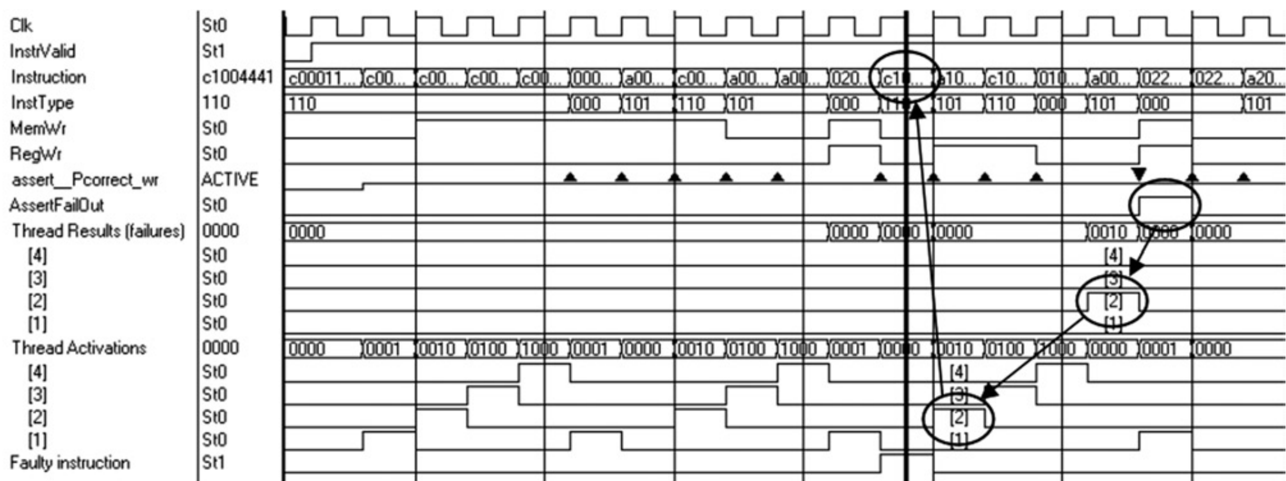

Fig. 9 *Using assertion threading to quickly locate the cause of an instruction execution error in the CPU pipeline example*

Table 1: Assertion-circuit resource usage in two MBAC modes

Assertion	Normal			Assertion completion		
	FF	LUT	MHz	FF	LUT	MHz
assert always {a&b} |−> {∼c; {d&∼c}[∗0:4]; c& ∼ d}; (Example 1)	6	8	433	6	**7**	**444**
assert always ({a} |=> {{c[∗0:1];d}|{e}}); (Example 2)	3	3	610	3	**2**	610
assert always ({a;b} |=> {c[∗0:1];d}); (Example 3)	4	3	611	4	3	611
assert always {a} |=> {{[∗2];b;∼c}|{[∗2];∼b;c}}; (Example 4)[a]	6	3	564	6	3	564
assert always {a} |=> {b;c;d;e}; (AMBA asr. [17])[a]	5	5	514	5	**4**	**611**
assert always {a;∼a} |=> {(∼a)[∗0:15];a} abort b; (AMBA asr. [7])[a]	18	**17**	**611**	18	23	312
assert always {a;b} |=> {c;{{d[∗];e}[+];f} && {g[∗]}} abort h; (PCI asr. [2])[a]	5	10	468	5	**7**	**470**
assert always {a} |=> {e;d;{b;e}[∗2:4];c;d};	15	21	329	15	**15**	**430**
assert always {a} |=> {b; {c[∗0:2]}|{d[∗0:2]} ; e};	7	12	333	7	**9**	**414**
assert always {{{b;c[∗1:2];d}[+]}: {b;{e[−>]};d}} |=> next a;	8	7	473	8	7	473
assert always {a} |=> {{{c[∗1:2];d}[+]} && {{e[−2:3]};d}};	16	38	304	16	**31**	**386**
assert always {a} |=> {{{b;c[∗1:2];d}[+]} & {b;{e[−2:3]};d}};	44	141	**260**	44	**139**	250
assert always {a} |=> {{{b;c[∗1:2];d}[+]} && {b;{e[−>2:3]};d}};	35	118	251	35	**100**	**281**

[a]Simplified Booleans

modes in [11]) can be observed by comparing the automata for both sequences appearing in the assertion in Fig. 2*b*. In this example, the occurrence and failure modes correspond to the left and right sides of the |=> operator, respectively.

To complete the threading, the sequence output is defined as the disjunction of the threaded automata outputs. Seen from the sub-circuit boundary, a multi-threaded sub-circuit's behaviour is identical to that of a non-threaded sub-circuit. Threading applies to any PSL property in which one or more sequences appear. Threading of a simple Boolean expression used at the property level is obviously not performed. A tradeoff is required between an accurate location of the source of the failure and hardware resources, as will be shown in Section 4.

An example scenario where assertion threading is useful is in the verification of highly pipelined circuits such as a CPU pipeline or a packet processor, where temporally complex sequences are modelled by assertions. In such cases, it is desirable to partition sequences into different threads in order to separate a failure sequence from other sequences. Once the sequence processing is temporally isolated, the exact cause of the failure can be more easily identified. The following case study shows how assertion threading can be used to quickly identify incorrect instruction executions in a CPU.

3.5.1 Assertion threading: CPU execution pipeline example: A simplified CPU execution pipeline, similar to the DLX [16] RISC CPU with five levels of pipeline, was coded in RTL and two classes of instructions are considered, namely memory writes and register writes. This CPU is used to execute instructions that contain memory and register manipulation. An error injection mechanism is also incorporated into the instruction decoder, such that errors can be inserted in the execution pipeline. Memory writes are committed at the fourth level (MEM stage) in the pipeline, and register writes are committed at the fifth level (WB stage) in the pipeline. For a given WRITE instruction, only a single destination is allowed by the architecture (Memory or Register).

Example 4 shows the PSL code used to create an assertion checker circuit for monitoring the memory write or register write instructions. The sequences '*Smemwr*' and '*Sregwr*' are built to ensure that a write is either to the register file or to external memory. In this CPU pipeline, those temporal expressions represent the same 'store' instruction

at two different pipeline stages. The sequence *Swr_instr* models the instruction decoder detecting the presence of a write instruction. The property *Pcorrect_wr* ensures that this write instruction will either result in a memory write or a register update (but never both).

Example 4: The assertion for a pipelined CPU write instruction in Fig. 8 is given to the MBAC Compiler along with the CPU RTL in Verilog.

The resulting checker is instantiated in the CPU architecture. The CPU along with its checker are exercised by a testbench running various verification scenarios.

Fig. 9 shows the resulting simulation trace. The dependency graph is used to determine the list of signals that relate to the assertion being debugged, and by extension the signals that need to be logged in the wave window. The AssertFailOut signal is asserted at a given time point, indicating a violation in the correctness of the write instruction behaviour. In this example, the instruction was committed to both the memory and the register file, which is impossible in this architecture. Tracing back through the Thread-Results vector, we find that thread #2 has detected the failure. Working back through the activations of this thread, it can be observed that the instruction causing the error is highlighted by the cursor in the figure. The assertion threading helps to isolate the source of the faulty sequence, and allows us to quickly determine which specific instruction was responsible for the assertion failure. In our example, for the sake of simplicity, the CPU executes one instruction per clock. In more complex problems, some instructions could take a variable number of cycles to execute; assertion threading would become an even more important asset to help debug these types of circuits.

4 Experimental results

The effects of assertion threading, assertion completion and activity monitors are explored by synthesising the assertion circuits produced by our checker generator using ISE 8.1.03i from Xilinx, for a XC2V1500-6 FPGA. The dependency graphs from Section 3.1 do not influence the circuits generated by the checker generator, while the assertion and coverage counters from Section 3.3 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.

Some of the assertions used in this section are from [2] and [17], while others were created during the development of MBAC to exercise the checker generator as thoroughly as possible. Typical assertions, such as most of the assertions used for verifying bus protocols, span few clock cycles and do not showcase the strength of our checker generator because they are easily handled. In the AMBA, PCI and CPU example assertions appearing in Tables 1–3, complex Boolean layer expressions are replaced by simplified Boolean symbols. As witnessed in Example 4, full assertions require much more space to write, and in their original form they are not practical for inclusion in Tables 1–3. Synthesising the full expressions does not change the temporal complexity of the automata; synthesising assertions with simple Boolean expressions allows us to better show the logic required for capturing the structure of an assertion in circuit form. This is a common practice when benchmarking checkers [10, 11, 18].

An upper bound on the area penalty of checkers in hardware is obtained assuming that no sharing of common circuit primitives takes place. Since the checkers monitor only the internal circuit signals, the extra loading can at worst add small delays, which can be kept low by following standard synthesis techniques. For instance, for the signals in the critical path that are monitored by an assertion, small buffers can be inserted to minimise the loading of the circuit under debug.

4.1 Assertion completion and activity monitoring

As described in Section 3.2, assertions can also be compiled in completion mode as opposed to the typical failure mode. Table 1 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 utilises slightly less combinational logic (LUTs),

Table 2: Resource usage of assertion circuits and activity monitors

<table>
<tr><th></th><th>FoCs</th><th></th><th></th><th>MBAC</th><th></th><th></th><th>MBAC + Act.Mon.</th><th></th><th></th></tr>
<tr><th>Assertion (assert *x*)</th><th>FF</th><th>LUT</th><th>MHz</th><th>FF</th><th>LUT</th><th>MHz</th><th>FF</th><th>LUT</th><th>MHz</th></tr>
<tr><td>always {a&b} |−> {∼c; {d&∼c}[∗0:4]; c&∼d}; (Example 1)</td><td>6</td><td>10</td><td>341</td><td>6</td><td>**8**</td><td>**433**</td><td>6</td><td>11</td><td>429</td></tr>
<tr><td>always ({a} |=> {{c[∗0:1];d}|{e}}); (Example 2)</td><td>3</td><td>3</td><td>610</td><td>3</td><td>3</td><td>610</td><td>3</td><td>4</td><td>610</td></tr>
<tr><td>always ({a;b} |=> {c[∗0:1];d}); (Example 3)</td><td>4</td><td>3</td><td>564</td><td>4</td><td>3</td><td>**611**</td><td>4</td><td>5</td><td>564</td></tr>
<tr><td>always {a} |=> {{[∗2];b; ∼ c}|{[∗2];∼b;c}}; (Example 4)[a]</td><td>6</td><td>3</td><td>564</td><td>6</td><td>3</td><td>564</td><td>6</td><td>5</td><td>559</td></tr>
<tr><td>always {a} |=> {b;c;d;e}; (AMBA asr. [17])[a]</td><td>5</td><td>**3**</td><td>417</td><td>5</td><td>5</td><td>**514**</td><td>5</td><td>6</td><td>509</td></tr>
<tr><td>always {a;∼a} |=> {(∼a)[∗0:15];a} abort b; (AMBA [17])[a]</td><td>N.O.</td><td></td><td></td><td>**18**</td><td>**17**</td><td>**611**</td><td>18</td><td>23</td><td>564</td></tr>
<tr><td>always {a;b} |=> {c;{{d[∗];e}[+];f} && {g[∗]}} abort h; [2][a]</td><td>N.O.</td><td></td><td></td><td>**5**</td><td>**10**</td><td>**468**</td><td>5</td><td>12</td><td>411</td></tr>
<tr><td>never {a;d;{b;a}[∗2:4];c;d};</td><td>25</td><td>23</td><td>564</td><td>**12**</td><td>**11**</td><td>564</td><td>12</td><td>15</td><td>559</td></tr>
<tr><td>always {a} |=> {e;d;{b;e}[∗2:4];c;d};</td><td>N.O.</td><td></td><td></td><td>**15**</td><td>**21**</td><td>**329**</td><td>15</td><td>26</td><td>312</td></tr>
<tr><td>always {a} |=> {b; {c[∗0:2]}|{d[∗0:2]} ; e};</td><td>7</td><td>**11**</td><td>333</td><td>7</td><td>12</td><td>333</td><td>7</td><td>14</td><td>331</td></tr>
<tr><td>never {{{b;c[∗1:2];d}[+]} && {b;{e 2:3]};d} };</td><td>36</td><td>44</td><td>394</td><td>**16**</td><td>**19**</td><td>**395**</td><td>16</td><td>24</td><td>381</td></tr>
<tr><td>always {a} |=> {{{c[∗1:2];d}[+]} && {{e[− > 2:3]};d}};</td><td>N.O.</td><td></td><td></td><td>**16**</td><td>**38**</td><td>**304**</td><td>17</td><td>40</td><td>293</td></tr>
<tr><td>always {a} |=> {{{b;c[∗1:2];d}[+]} & {b;{e[− > 2:3]};d}};</td><td>N.O.</td><td></td><td></td><td>**44**</td><td>**141**</td><td>**260**</td><td>44</td><td>150</td><td>259</td></tr>
<tr><td>always {a} | = > {{{b;c[∗1:2];d}[+]} && {b;{e[− > 2:3]};d}};</td><td>N.O.</td><td></td><td></td><td>**35**</td><td>**118**</td><td>**251**</td><td>35</td><td>128</td><td>243</td></tr>
</table>

N.O., no output
[a]Simplified Booleans

Table 3: Area tradeoff metrics for assertion threading

Assertion	None			Two-way			Four-way			Eight-way		
	FF	LUT	MHz	FF	LUT	MHz	FF	LUT	MHz	FF	LUT	MHz
A1	**6**	**8**	**433**	15	18	386	29	33	306	57	62	241
A2	**12**	**11**	564	25	23	564	49	46	433	97	91	408
A3	**15**	**21**	**329**	33	44	298	65	83	252	129	164	235
A4	**16**	**19**	395	33	39	395	65	77	395	129	160	318
A5	**26**	**80**	246	57	165	**252**	113	297	205	225	570	177
A6	**35**	**118**	**251**	73	235	239	145	430	213	289	881	186
A7	**6**	**3**	**564**	15	11	442	29	20	362	not required		
A8	**5**	**5**	**514**	13	16	323	25	24	326	not required		
A9	**18**	**17**	611	39	38	442	77	75	364	153	144	297
A10	**5**	**10**	**468**	13	23	311	25	39	278	49	67	235

A1: assert always {a&b} |−> {∼c; {d&∼c}[∗0:4]; c&∼d}; (Example 1)
A2: assert never {a;d;{b;a}[∗2:4];c;d};
A3: assert always {a} |=> {e;d;{b;e}[∗2:4];c;d};
A4: assert never { {{b;c[∗1:2];d}[+]} && {b;{e[− > 2:3]};d} };
A5: assert always {a} |=> {{{b;c[∗1:2];d}[+]} : {b;{e[−>]};d}};
A6: assert always {a} |=> { {{b;c[∗1:2];d}[+]} && {b;{e[− > 2:3]};d} };
A7: assert always {a} |=> {{[∗2];b;∼c}|{[∗2];∼b;c}}; (Example 4)[a]
A8: assert always {a} |=> {b;c;d;e}; (AMBA asr. [17])[a]
A9: assert always {a;∼a} |=> {(∼a)[∗0:15];a} abort b; (AMBA asr. [17])[a]
A10: assert always {a;b} |=> {c;{{d[∗];e}[+];f} && {g[∗]}} abort h; (PCI asr. [2])[a]

[a]Simplified Booleans

and runs slightly faster than its normal-mode version (i.e. regular failure matching).

The activity monitors introduced in Section 3.4 are used to observe when sequences are undertaking a matching. An activity signal is composed of the disjunction of state signals from all of the states in a given automaton, as witnessed in Fig. 2. Table 2 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 FF 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 [9, 19].

4.2 *Assertion threading*

As explained in Section 3.5, assertion threading replicates sequence circuits in order for the failure conditions to be isolated from other activations. This was shown to ease the debugging process considerably, particularly when temporally complex assertions are used. Table 3 shows how the resource utilisation scales as a function of the number of hardware threads. The threaded assertion circuit used in the example in Fig. 9, with full Boolean expressions, actually synthesises to 29 FFs, 21 LUTs, with a maximum frequency of 362 MHz, which corresponds to virtually the same metrics as the simplified version used in test case A7 (four-way column).

Because an eight-way threading is only useful for sequences that span at least eight clock cycles, the assertions used must have a certain amount of temporal complexity for the results to be meaningful. Since the assertion from Example 4 (A7) and the AMBA assertion in A8 both contain simple left-sides for |=>, along with right-side sequences that span four clock cycles, they do not benefit from eight-way assertion threading. As we expected, the experimental data show that the resource utilisation scales linearly with the number of hardware threads.

5 Conclusion

In this paper we have presented techniques that facilitate debugging within the assertion-based framework, either in the emulation or in the silicon debug stages. By selecting various features, debugging is enhanced by providing better visibility, 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.

6 References

1 Abramovici, M., Bradley, P., Dwarakanath, K., Levin, P., Memmi, G., and Miller, D.: ‘A reconfigurable design-for-debug infrastructure for SoCs’. In Proc. 43rd Design Automation Conference (43rd DAC), 2006, pp. 7–12
2 Foster, H., Krolnik, A., and Lacey, D.: ‘Assertion-based design’ (Kluwer Academic Publishers, Norwell, MA, 2004, 2nd edn.)
3 Boule, M., and Zilic, Z.: ‘Incorporating efficient assertion checkers into hardware emulation’. Proc. 23rd IEEE Int. Conf. Computer Design (ICCD’05), 2005, pp. 221–228
4 Hsu, Y.-C., Tabbara, B., Chen, Y.-A., and Tsai, F.: ‘Advanced techniques for RTL debugging’. Proc. 40th Design Automation Conf. (40th DAC), 2003, pp. 362–367
5 Hsu, Y.-C., Tsai, F., Jong, W., and Chang, Y.-T.: ‘Visibility enhancement for silicon debug’. Proc. 43rd Design Automation Conf. (43rd DAC), 2006, pp. 13–18
6 Boulé, M., Chenard, J.S., and Zilic, Z.: ‘Adding debug enhancements to assertion checkers for hardware emulation and silicon debug’. Proc. 24th IEEE Int. Conf. on Computer Design (ICCD’06), 2006, pp. 294–299

7 Accellera Organization, Inc.: 'Property specification language – reference manual, v.1.1'. available at: www.eda.org/vfv/docs/PSL-v1.1.pdf, 2004
8 Eisner, C., and Fisman, D.: 'A Practical Introduction to PSL' (Springer-Verlag, New York, NY, 2006)
9 Abarbanel, Y., Beer, I., Glushovsky, L., Keidar, S., and Wolfsthal, Y.: 'FoCs: automatic generation of simulation checkers from formal specifications'. Proc. 12th Int. Con. on Computer Aided Verification (CAV'00), 2000, pp. 538–542
10 Boulé, M., and Zilic, Z.: 'Efficient automata-based assertion-checker synthesis of psl properties'. Proc. 2006 IEEE Int. High Level Design Validation and Test Workshop (HLDVT'06), 2006, pp. 69–76
11 Boulé, M., and Zilic, Z.: 'Efficient Automata-Based Assertion-Checker Synthesis of SEREs for Hardware Emulation'. Proc. 12th Asia and South Pacific Design Automation Conf. (ASP-DAC2007), 2007, pp. 324–329
12 Sidhu, R., and Prasanna, V.: 'Fast regular expression matching using FPGAs'. Proc. 9th Annual IEEE Symp. on Field Programmable Custom Computing Machines (FCCM'01), 2001, pp. 227–238
13 Hopcroft, J., Motwani, R., and Ullman, J.: 'Introduction to automata theory, languages and computation' (Addison-Wesley, 2000, 2nd edn.)
14 Beer, I., Ben-David, S., Eisner, C., and Rodeh, Y.: 'Efficient detection of vacuity in temporal model checking', *Formal Methods Syst. Design*, 2001, **18**, (2), pp. 141–163
15 Ruah, S., Fisman, D., and Ben-David, S.: 'Automata construction for on-the-fly model checking PSL safety simple subset', Technical Report H-0234, IBM, 2005
16 Hennessy, J.L., and Patterson, D.V.: 'Computer architecture: a quantitative approach' (Morgan Kaufmann Series in Computer Architecture and Design, San Francisco, CA, 2003, 3rd edn.)
17 Cohen, B., Venkataramanan, S., and Kumari, A.: 'Using PSL/ sugar for formal and dynamic verification' (VhdlCohen Publishing, Los Angeles, CA, 2004)
18 Borrione, D., Liu, M., Morin-Allory, K., Ostier, P., and Fesquet, L.: 'On-line assertion-based verification with proven correct monitors'. Proc. 3rd ITI Int. Conf. Information and Communications Technology (ICICT 2005), 2005, pp. 123–143
19 IBM Alpha Works: 'FoCs property checkers generator', version 2.03. available at: www.alphaworks.ibm.com/tech/FoCs, 2006