Assertion Checkers in Verification, Silicon Debug and In-Field Diagnosis

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

Abstract

Assertion Based Design, and more specifically, Assertion Based Verification (ABV) is quickly gaining wide acceptance in the design community. Assertions are mainly targeted at functional verification during the design and verification phases. In this paper, we concentrate on the use of assertions in post-fabrication silicon debug. We develop tools that efficiently generate the checkers from assertions, for their inclusion in the debug phase. We also detail how a checker generator can be used as a means of circuit design for certain portions of self test circuits, and more generally the design of monitoring circuits. Efficient subset partitioning of checkers for a dedicated fixed-size reprogrammable logic area is developed for efficient use of dedicated debug hardware.

1. Introduction

Hardware verification aims to ensure that a design fulfills its given specification by either formal or dynamic (simulation based) techniques. As one facet of Assertion-Based Design, Assertion-Based Verification (ABV) is quickly emerging as the dominant methodology for performing hardware verification in practice. Using temporal logic, a precise description of the expected behavior of a design is modeled, and any deviation from this expected behavior is captured by simulation or by formal methods. Hardware assertions are typically written in a verification language such as PSL (Property Specification Language, IEEE 1850 standard) or SVA (SystemVerilog Assertions). When used in dynamic verification, a simulator monitors the Device Under Verification (DUV) and reports when assertions are violated. Information on where and when assertions fail is an important aid in the debugging process, and is the fundamental reasoning behind the ABV methodology.

Assertion-Based Design practices also advocate the use of assertions as part of the design effort, when used as a formal specification (describing designer intent). Assertions are also applied beyond design and verification, when used with a checker generator. In such cases, hardware checkers can be produced to create permanent circuitry that can be added to the design in order to perform self-test, on-line silicon monitoring and diagnosis assistance during the lifespan of the IC.

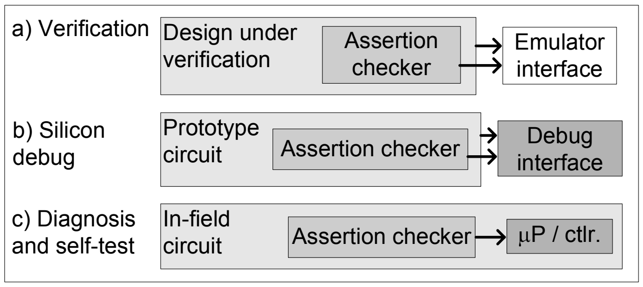

Figure 1. Usage scenarios for hardware assertion checkers.

Individual assertions, once converted into circuit form, are also referred to as assertion circuits. These checker circuits are typically expressed in a hardware description language. An assertion checker is a circuit that captures the behavior of a given assertion, and can be included in the DUV for in-circuit assertion monitoring. Efficient circuit-level assertion checkers generated by our tool can be used in a variety of contexts, three of which are outlined in Fig. 1. These contexts are: a) checkers are used in the verification stage when the design is to be simulated, emulated in hardware or executed on a simulation accelerator; b) checkers are used to perform post-fabrication in-circuit debugging; c) in-field custom circuits for in-chip diagnosis and self test. The circuits may be temporary in verification, whereas in contexts b) and c), they represent a permanent addition to the IC. Assertion checkers for verification are the topic of extensive recent research; in this paper, we focus on the remaining two contexts. The contributions of this paper are:

- Introduction of resource-efficient assertion checkers for use in post fabrication silicon debug;
- Use of the concept of a checker generator for performing synthesis of self-test and diagnosis circuits;
- Algorithm for assertion partitioning in the presence of a reprogrammable circuit area dedicated for in-circuit run-time execution of checker circuits;
- Use of above techniques for on-line and in-field monitoring and diagnosis of circuits.

In the following section, we give a short background on assertions, assertion checkers and checker generators. In

Section 3, we show how assertion checkers can be used beyond the verification stages, and into full silicon assertion debugging. Section 3.1 shows how assertions, combined with a checker generator can be used to automatically design certain types of circuits. Example scenarios are shown in self-test and in-field diagnosis. Section 4 shows how assertion checkers can be managed in a dedicated reprogrammable logic area, for use in system-on-chip designs as an example. Experimental results for this section are reported in Section 5.

2. Background

Assertions are at the foundation of assertion based verification, and are specified using a variety of Boolean expressions as primitives, along with regular expressions and numerous temporal operators. Assertion languages have complex syntax and semantics and are beyond the scope of this paper; however, to illustrate how assertion checkers can be generated, an example assertion in PSL is shown below.

$$\textsf{assert always}\ (\{\textsf{rose}(req)\}\ |\!\!=\!> \{req[*0{:}4]\,;req\,\&\,grant\});$$

In this example, the arbiter is expected to grant the bus to the client within four clock cycles of the request signal going high. The client must also keep its request signal active until it receives control of the bus which is indicated by the $grant$ signal. If any of these post-conditions do not occur, the assertion will trigger and indicate an error. The |=> operator is a temporal implication, with preconditions and postconditions appearing as left and right arguments respectively. rose(b) is an operator that evaluates to true when the Boolean expression b is true in the current clock cycle, and was false in the previous cycle. In this example, the post-condition is a regular expression consisting of a temporal concatenation ";" of two sub-expressions, the left of which contains a repetition range and the right expression is Boolean.

Assertions are used at various stages of the design process. In tools such as Modelsim and VCS, when a circuit is simulated the assertions are monitored by the simulation kernel. When designs are to be emulated in hardware, assertions can not be directly mapped into the hardware because they are written in a higher-level language that is not necessarily amenable to synthesis. When the power of assertions is to be used in hardware, a *checker generator* is used to automatically produce monitoring circuits (also called *assertion checkers*), from the given assertion statements. Two such checker generators are MBAC [6, 7] and FoCs from IBM [1]. The MBAC checker generator creates resource efficient assertion checker circuits, and supports the entire simulatable subset of PSL. These checkers can also be instrumented with various debugging enhancements [5].

Running MBAC on the above assertion creates an assertion checker circuit comprising 7 flip-flops and 8 four-input combinational logic cells. The circuit monitors the design signals, and produces a single-bit output that indicates the status of the assertion in real-time. If the assertion signal remains low during the entire execution, the design is found to respect the given property, provided sufficient stimulus and coverage were exercised. This checker can then be instantiated in the device under test, in order to perform verification in hardware emulation, or to perform debugging of fabricated silicon, or even to perform in-field run-time diagnosis.

Assertions and debugging are receiving attention from many EDA companies. Temento's DiaLite product accepts assertions and provides in-circuit debugging features. DAFCA's ClearBlue solution [3] offers silicon debugging instruments such as in-circuit trace buffers for capturing signals or supplying vectors, signal probe multiplexers and logic analyzer circuitry. Assertions can also be instrumented and changed dynamically in specialized reprogrammable logic. The idea of assertion grouping was brought up in [3], and is explored further in this paper in Section 4.

3. Checkers for Silicon Debug

The silicon debugging process is aimed at finding and possibly correcting design errors in a post-fabricated IC, usually referred to as *first silicon*. Assertion checkers produced by MBAC can not only be used for emulation and simulation verification before fabrication, but also for post-fabrication, when a set of assertion checkers is purposely left in the design. The checkers can test for functional faults and timing issues which can not be fully tested pre-fabrication. By connecting the checker outputs to the proper external equipment or on-board read-back circuits, the user can get immediate feedback on assertion failures in order to start the debugging process. A checker generator capable of producing resource-efficient checkers is clearly an advantage when checkers take up valuable area that has to be committed before tape-out.

Assertion-based silicon debug differentiates itself from emulation based verification because in silicon debug, the design is implemented in its intended technology, as opposed to being implemented in reprogrammable logic during hardware emulation. This allows at-speed debugging under expected operating conditions, and assertion checkers play an important role here as well. Figure 2 a) shows how assertion checkers in silicon are used to monitor the state of the device under test during the entire execution. This monitoring mode is identical to that which is used verification, with the nuance that the checkers exist in permanent silicon and can be used during the lifetime of the device, as opposed to temporary verification checkers which are removed before tape-out.

3.1. Checkers in Self-Test and In-Field Diagnosis

The checkers for silicon debugging mentioned at above serve their purpose, but can ultimately be removed for production quantity re-spins. In a more general usage scenario, the expressive power of assertions, combined with a checker

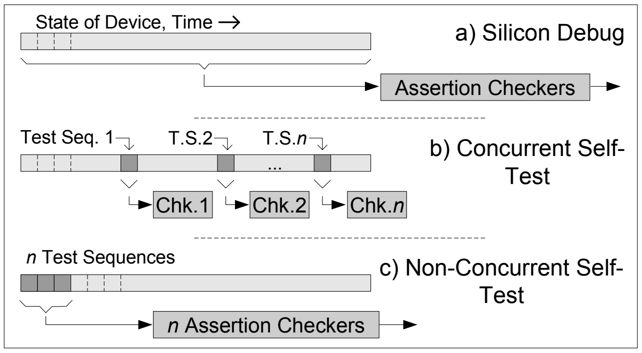

Figure 2. Debugging and self-test using checkers.

generator can be used to actually perform explicit circuit design, going beyond the bounds of verification and debugging. This is not unlike the Production Based Specification research [11], which was based on regular expressions. In our proposed scenario, any form of monitoring circuit that can be expressed by an assertion, once fed into our checker generator, can produce a complex error-free circuit instantly. These circuit-level checkers are in fact more akin to actual design modules rather than verification modules.

A checker generator allows the flexibility of automatically generating custom monitor circuits from any assertion statement. 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. Using assertions and a checker generator can be a very efficient way of automating the design of certain types of circuits. An example where this technique can be utilized is in designing certain portions of self-test circuits (and Built-In Self Test [2]). Off-line BIST techniques are well established, and are based on the traditional TPG → CUT → ORA architecture shown in Fig. 3 a). (TPG = Test Pattern Generator, CUT = Circuit Under Test, ORA = Output Response Analysis). Off-line BIST techniques typically employ a mixture of pseudo-random and deterministic TPG.

Our technique based on assertions also applies to self test, albeit at a higher level. Test pattern generation is instead referred-to as test sequence generation (TSG). Figure 3 b) shows an assertion-based off-line self-test architecture, whereby test sequences are applied to the input of the CUT, and assertion checkers are used as the response analysis. In this approach the signature can be encoded as one bit, representing success or failure. The offline self-test, when executed prior to device startup is considered non-concurrent, and is illustrated in Figure 2 c). The use of assertions and a checker generator allows the response analysis circuitry to be designed with greater ease.

Our checker-based self-test techniques also apply to the design of on-line self-test circuits [4], as shown in Fig. 2 b). In this scenario, the checker generator is used to design the analysis circuits that correspond to the given test sequences.

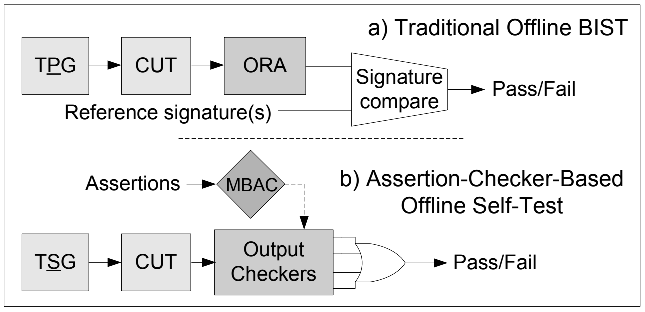

Figure 3. Traditional BIST vs. self-test using checkers.

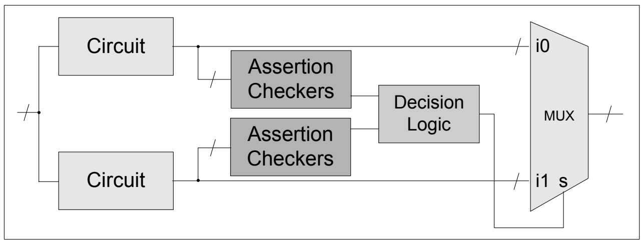

Figure 4. Run time diagnosis using assertion checkers for redundancy control.

Contrary to silicon debug and the other self-test techniques mentioned previously, a checker for a given test sequence is only used as a response analyzer when the test sequence is being exercised. In the concurrent self-test model, the device is momentarily interrupted for testing, or alternately, unused resources are concurrently tested during runtime.

Using assertions and a checker generator as a means of circuit design poses difficulties when it comes to stimulus generation; however, the design of many types of monitoring and analysis circuitry can benefit directly from this technique. The high-level expressiveness of an assertion language combined with an assertion compiler can be used as a quick method to automatically design circuits.

If checkers are incorporated in the final circuit design, in-circuit diagnostic routines can be implemented during field deployment. Assertion checkers can be an integral part of any design which attempts to assess its operating conditions on-line in real time. Run-time assertion checker monitoring can be performed, and the results of checkers analyzed by an on-board CPU which can then send regular status updates off-chip. Real-time diagnosis based on in-circuit checkers can be especially important in mission-critical environments. For example, if a multitude of assertion checkers running concurrently with the device were to detect an error, a secondary redundant system could be instantly activated. Figure 4 shows an example of how our methodology can be used to design the diagnosis circuits for switching in redundant systems. Designing an array of safety-checking circuits can be more easily performed using assertions and a checker generator for circuit design.

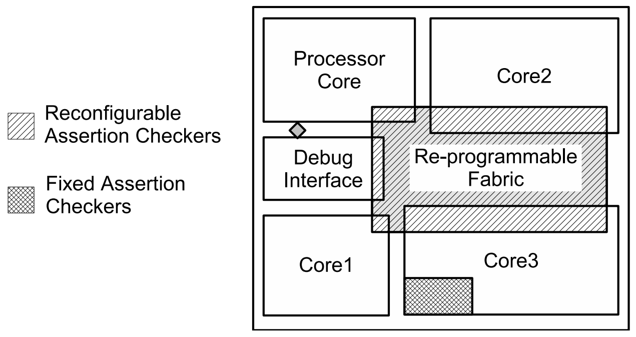

Figure 5. Typical SoC floorplan implementing fixed and re-programmable assertion checkers.

4. Dedicated Programmable Logic for Checkers

Hardware assertion checkers can be incorporated as a part of the final silicon as dedicated checkers that continuously monitor the circuit for abnormal conditions. In a typical ASIC, some of the IP cores are known to be quite robust from previous use or because they are provided by a third party vendor with previous successful tape-outs. Therefore, a balance between risk mitigation and on-chip assertion capabilities has to be calculated. Programmable-logic fabric or reconfigurable elements are increasingly inserted into ASICs to allow corrections of silicon bugs or to bypass faulty modules. Since this reconfigurable fabric should be unused at the initial tape-out, it represents an excellent opportunity to include assertion monitors with no cost impact.

Figure 5 shows different levels of core confidence, as could be encountered in a typical System-on-Chip (SoC) design, for example. Core1 could have been used in a previous design, thus the confidence is high and a limited number of connectivity points are shared with the programmable fabric. Core3 could be a new design and thus being more risky, more re-programmable resources are dedicated to potential bug fixing, while additional checkers can be built into the silicon as extra precaution and to assist post-silicon debug.

The assertion checkers benefit from observability on the main system buses for protocol checking and assertion-based debugging enhancements. Before tape-out, an analysis of each checker circuit is performed and the routing overhead is estimated based on each assertion's input dependencies. Once the design is locked with a specified list of available monitoring points, the tool can provide the designer with all the assertion checkers that will be supported by the future ASIC. New assertion checkers can be generated after tape-out as long as they respect the silicon constraints. The reprogrammable logic IP core can even be combined with the assertion based concurrent BIST from Fig. 2 c). In this scenario, the microprocessor can coordinate the instantiation of the proper checkers for each test sequence in the reprogrammable fabric. Checker groups (also called partitions, or subsets) are instantiated one after the other in the reconfigurable area, to correspond with the set of test sequences being run. Reprogramming FPGA fabric on the fly for different tasks is known as run-time reconfiguration [10].

4.1. Assertion Checker Partitioning Algorithm

The MBAC checker generator can process any set of properly constructed PSL statements (no branching time logics) and transform them into synthesizable RTL code. Since the resulting code is heavily instrumented through comment blocks, it contains the necessary information to perform higher-level analysis of the checker integration. A second tool builds a database of each of the checker modules by automating their synthesis and extracting all the relevant metrics. In our current implementation, we use the Xilinx XST synthesis tools for VirtexII FPGA devices.

Once the checkers have been individually synthesized and their sizing metrics are obtained, the partitioning algorithm shown in Figure 6 is used to create subsets of checkers suitable for multiple reconfigurations in the reprogrammable logic area. This algorithm is based on solving the subset-sum problem by dynamic programming [9]. However, because the circuit metrics comprise two variables, namely # of flip-flops (FF) and # of lookup tables (LUT), the typical subset-sum procedure can not be employed directly on its own. We have therefore developed a two-phase algorithm, which returns a near-optimal partition, given the circuits' metrics and the size of the reprogrammable area (also specified as # of flip-flops and # of lookup tables).

Phase 1 in the algorithm (lines 3-8) uses flip-flops as the dominant metric and performs subset-sum on this metric (line 5). The subset-sum algorithm requires that the circuits be sorted in increasing order according to the dominant metric (line 4). A search is then performed for the best subset according to the size limit of this dominant metric which also respects the maximum size for the secondary metric (line 6). Once the best subset has been determined, it is logged and removed from the set (lines 7 and 8). This procedure continues until the set of checkers is empty (line 3).

The dominant / secondary metrics are interchanged and the same procedure is repeated (lines 9 to 14). A comparison is then made between both phases (lines 15 to 18), and the solution with the fewest subsets is logged. When both phases have the same number of subsets, it was empirically observed that the more balanced partition is the one for which the dominant metric corresponds to the metric which is the most constrained by the area limits (smallest freedom).

It can be shown by counterexample that the algorithm is not guaranteed to create an optimal partition; however, our experiments show that it drastically outperforms the brute force approach in computation time. Furthermore, when one of the metrics has a large amount of freedom with respect to its constraint, the problem tends toward a single variable subset sum for which our algorithm is optimal.

```
1: FUNCTION: SUBSET-CIRCUIT(set $C$ of circuit metrics (FF, LUT), $area_{FF}$, $area_{LUT}$)
2: $D \leftarrow C$
3: **while** there are circuits left in $C$ **do**    // phase 1 (dominant metric is #FFs)
4:     **sort** circuits $C$ according to $\#FF$s
5:     **build** dynamic programming table $T$ for subset-sum on $\#FF$s
6:     **search** $T$ for best subset $S$ such that $\sum_{s_i \in S} \#LUTs(s_i) < area_{LUT}$
7:     **log** subset circuits in $S$ as a group in phase 1 results
8:     **remove** circuits $S$ from $C$
9: **while** there are circuits left in $D$ **do**    // phase 2 (dominant metric is #LUTs)
10:    **sort** circuits $D$ according to $\#LUT$s
11:    **build** dynamic programming table $T$ for subset-sum on $\#LUT$s
12:    **search** $T$ for best subset $S$ such that $\sum_{s_i \in S} \#FFs(s_i) < area_{FF}$
13:    **log** subset circuits in $S$ as a group in phase 2 results
14:    **remove** circuits $S$ from $D$
15: **if** number of subsets in both phases differs **then**    // analysis
16:    **return** results of phase which has the fewest subsets (groups)
17: **else**
18:    **return** results of phase for which the subset-sum was performed on metric with smallest freedom
```

Figure 6. Assertion circuit partitioning algorithm.

5. Experimental Results

In this section we demonstrate the use of our algorithms. The MBAC checker generator is used to produce assertion checkers for two suites of assertions. The assertions are used to verify an AMBA slave device and AMBA AHB interface compliance, and were taken from Chapter 8 in [8]. Because of the temporal nature of the assertions, the assertion checkers utilize more combinational cells than flip-flops. However, the partitioning algorithm can operate on any type of circuits whether they are balanced or biased towards either flip-flops or combinational logic.

Table 2 shows the individual resource usage of checkers for the assertions in the AHB and mem_slave examples. In the table, N.A. means Not Applicable, and occurs for circuits containing only one FF with no feedback path (the MHz is a clk-to-clk figure). The checker generator used is MBAC version 1.71, and the checkers are synthesized with Xilinx XST 8.1.03i. The target device is an XC2V1500-6, and the synthesis is optimized for speed (as opposed to area). Table 1 shows how the assertion circuits from Table 2 are partitioned into a minimal number of sets by the subset-circuit algorithm, for a target area of 50 FFs and 50 four-input LUTs. In both cases, phase two results were logged (dominant LUTs). The right-most column lists the sums of the circuit metrics in each group.

Table 3 shows how the actual resource usage can be slightly diminished when the circuits that form a subset are actually synthesized together. As a general result, it can be expected that as the number of circuits per subset increases, the resource sharing becomes more important, and the overall metrics for a given subset become smaller. For comparison purposes, Table 3 also lists the full-set metrics, which are obtained by synthesizing all checkers as a single module.

Table 1. Checker partitions for reprogrammable area.

AHB example:

Subset	Assertion circuits in partition	ΣFF, ΣLUT
#1	{A9, A14, A15}	4, 50
#2	{A8, A22, A23, A25}	5, 50
#3	{A7, A10, A21, A24}	5, 50
#4	{A6, A11, A13}	6, 50
#5	{A1, A2, A3, A4, A5, A12, A16}	14, 50
#6	{A17, A18, A19, A20, A26}	29, 26
	Total:	63, 276

mem_slave example:

Subset	Assertion circuits in partition	ΣFF, ΣLUT
#1	{A6, A8, A19, A20, A21, A22, A26}	7, 50
#2	{A1, A11, A15, A18, A23, A24}	15, 50
#3	{A2, A3, A5, A7, A9, A10, A14, A16, A17, A25}	21, 50
#4	{A4, A12, A13}	6, 8
	Total:	49, 158

The end result is an efficient partition of checkers which minimizes the number of times the reprogrammable logic area must be reconfigured. A test procedure can then run a batch of test sequences with a given subset of checkers, then instantiate a new set of checkers, re-run the test sequences, and so forth. Once the verification with checkers is finished, the reprogrammable fabric can be used for the functionality of the intended design.

6. Conclusions

As assertion based verification becomes more dominant in the design community, hardware assertions can be used

Table 2. Resource usage of assertion checkers.

AHB example:

Assertion	FFs	LUTs	MHz	Assertion	FFs	LUTs	MHz	Assertion	FFs	LUTs	MHz
ahb_A1	2	2	667	ahb_A10	1	6	N.A.	ahb_A19	3	3	667
ahb_A2	2	3	611	ahb_A11	2	30	667	ahb_A20	3	2	667
ahb_A3	2	3	667	ahb_A12	2	18	667	ahb_A21	1	23	N.A.
ahb_A4	2	2	611	ahb_A13	2	18	611	ahb_A22	1	21	N.A.
ahb_A5	2	2	667	ahb_A14	1	12	N.A.	ahb_A23	1	21	N.A.
ahb_A6	2	2	667	ahb_A15	1	36	N.A.	ahb_A24	1	19	N.A.
ahb_A7	2	2	667	ahb_A16	2	20	667	ahb_A25	1	6	N.A.
ahb_A8	2	2	667	ahb_A17	2	2	611	ahb_A26	18	17	611
ahb_A9	2	2	667	ahb_A18	3	2	667				

mem_slave example:

Assertion	FFs	LUTs	MHz	Assertion	FFs	LUTs	MHz	Assertion	FFs	LUTs	MHz
mem_slave_A1	1	4	N.A.	mem_slave_A10	5	16	456	mem_slave_A19	1	5	N.A.
mem_slave_A2	2	4	667	mem_slave_A11	5	22	469	mem_slave_A20	1	6	N.A.
mem_slave_A3	2	2	667	mem_slave_A12	2	3	667	mem_slave_A21	1	6	N.A.
mem_slave_A4	2	2	667	mem_slave_A13	2	3	667	mem_slave_A22	1	1	N.A.
mem_slave_A5	1	2	N.A.	mem_slave_A14	2	3	667	mem_slave_A23	2	5	667
mem_slave_A6	1	7	N.A.	mem_slave_A15	2	3	667	mem_slave_A24	1	4	N.A.
mem_slave_A7	1	2	N.A.	mem_slave_A16	2	7	667	mem_slave_A25	1	3	N.A.
mem_slave_A8	1	7	N.A.	mem_slave_A17	1	2	N.A.	mem_slave_A26	1	18	N.A.
mem_slave_A9	4	9	417	mem_slave_A18	4	12	442				

Table 3. Subset and full-set synthesis.

AHB example:

Subset	FFs, LUTs
#1	4, 50
#2	5, 50
#3	5, 49
#4	6, 34
#5	13, 48
#6	29, 26
Total:	62, 257

Full-Set (FFs, LUTs)
60, 250

mem_slave example:

Subset	FFs, LUTs
#1	7, 43
#2	15, 47
#3	21, 47
#4	6, 8
Total:	49, 145

Full-Set (FFs, LUTs)
48, 129

beyond their intended purpose of pre-tape-out hardware verification. In this paper, we have shown how assertion checkers can be used for post-silicon debugging, and even for certain types of circuit design, with applications in built-in self test and in-field diagnosis. With the advent of system-on-chip designs, checkers can be instantiated in reprogrammable logic cores in the device at no cost before the logic is used for bug fixes. For such applications, a subset-partitioning algorithm was developed to optimize the debug resource usage, or in situations when the test-suite of assertion checkers can not fit in the reprogrammable area. This algorithm facilitates a low-overhead on-line concurrent self-test strategy and improved debugging.

References

[1] Y. Abarbanel et. al. FoCs: Automatic Generation of Simulation Checkers from Formal Specifications. In *12th Intl. Conf. on Computer Aided Verification*, pages 538–542, 2000.

[2] M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing & Testable Design*. Wiley-IEEE Press, 1994.

[3] M. Abramovici et. al. A Reconfigurable Design-for-Debug Infrastructure for SoCs. In *Proc. of the 43rd Design Automation Conference (43rd DAC)*, pages 7–12, 2006.

[4] H. Al-Asaad, B. Murray, and J. Hayes. Online BIST for Embedded Systems. *IEEE Design & Test of Computers*, 15(4):17–24, 1998.

[5] M. Boulé, J. Chenard, and Z. Zilic. Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug. *IEEE Intl. Conference on Computer Design (ICCD'06)*, pages 294–299, 2006.

[6] M. Boulé and Z. Zilic. Incorporating Efficient Assertion Checkers into Hardware Emulation. *IEEE Intl. Conference on Computer Design (ICCD'05)*, pages 221–228, 2005.

[7] M. Boulé and Z. Zilic. Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties. *IEEE Intl. High Level Design Validation and Test Workshop*, pages 69–76, 2006.

[8] B. Cohen, S. Venkataramanan, and A. Kumari. *Using PSL/Sugar for Formal and Dynamic Verification*. VhdlCohen Publishing, Los Angeles, California, 2004.

[9] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, 1999.

[10] M. Platzner. Reconfigurable Computer Architectures. http://citeseer.ist.psu.edu/490784.html.

[11] A. Seawright and F. Brewer. Clairvoyant: A Synthesis System for Production-Based Specification. *IEEE Transactions on VLSI Systems*, 2(2):172–185, 1994.