Assertion-based Verification for the SpaceCAKE Multiprocessor – A Case Study

Milind Kulkarni

Benita Bommi J

Philips Research India
#1, Murphy Road, Ulsoor, Bangalore, India
Milind.Kulkarni@philips.com
Benitabommi@rediffmail.com

Abstract. This paper presents a case study of the application of assertion-based verification to a multi-million-gate design of the SpaceCAKE architecture with shared L2 cache. SpaceCAKE L2 cache is highly configurable and implements Distributed Shared Memory (DSM) architecture. This paper discusses the issues faced during the functional verification of this architecture. A number of techniques are employed to verify the design. The paper serves as a case study for verification of such a complex architecture. A description of the different techniques that were used to verify this architecture and an assessment of using a comprehensive coverage-driven verification plan that exploits the benefits of the traditional simulation techniques through the use of assertions is presented. We have found that the tools, currently provided by the market, for assertion-based static verification approach need more maturity. A 50% reduction in debug time has been achieved through the use of assertions.

1. Introduction

Verification of SoCs through self-checking testbenches, directed tests, regression tests and simulation-based verification are the order of the day. The effort required to implement the above mentioned techniques of verification for a multi-million gate is monumental. In spite of the huge time spent on verifying the design using these techniques, the ability of the techniques to uncover the source of error is limited. The ability to present sources of errors is very necessary while verifying massive designs, as it is practically impossible to locate the source of error in such designs with reasonable effort. This necessity paved the way for implementing assertion-based verification for this design.

The paper discusses the areas where assertions can be applied without increasing the

simulation time overhead to a prohibitive level while still facilitating verification of the complete functionality of the design. It also discusses the use of assertions in acquiring coverage information that identify functionalities that have not been verified.

The organization of the paper is as follows. Section 2 is a description of the architecture verified. Section 3 discusses the need of assertion-based verification for multi-million gate designs. Section 4 is an account of the process of identification and implementation of the assertions.

Section 5 is a note on the implemented static verification of assertions. Section 6 is a description of the test set-up used to dynamically verify this architecture. Section 7 is a discussion of the results obtained through the use of this methodology followed by the conclusion in section 8.

2. The SpaceCAKE architecture

SpaceCAKE is a homogenous, tile-based, DSM architecture [1][2]. The tile is a multiprocessor system on a chip consisting of a number of programmable processor cores with an integrated L1 caches with Harvard architecture (i.e., separated instruction and data caches), a snooping bus-based interconnection network (ICN) which connects the cores, L2 cache, and the memory management unit interface to the off-chip DDR memory as shown in Fig.1. The tiles communicate via high-speed routers. Cache coherence within the tile and across tiles is implemented so that synchronization is taken out of software programmers botheration.

The features of this architecture are as follows:

- Distributed Shared Memory architecture.
- Level-1 and Level-2 Cache coherency.
- Shared resource reservation framework.
- Software controlled cache operations and debug functionality.
- Performance measurement based run-time adaptations.
- Multiple outstanding requests with optional out of order processing.

The interconnection network and the processor cores support hardware cache coherence, which greatly simplifies the programming model. There are multiple memory banks to increase the concurrency and improve throughput. Since the snooping interconnection network supports concurrent transactions, multiple CPUs can be busy with memory transactions at the same time, as long as they are accessing different memory banks. The blocks labeled SPF represent the special purpose hardware functions that are key to the computational efficiency of SpaceCAKE chip. All SPFs execute under control of a task running on any of the processors. Inter-tile communication takes place

through the routers. The interconnect network consists of a set of multi-purpose registers which can be software configured to implement Distributed Shared Memory (DSM) functionality. Due to the multiple configurations possible, high concurrency and advanced features with respect to cache coherency and snooping protocols the verification effort required is considerably high.

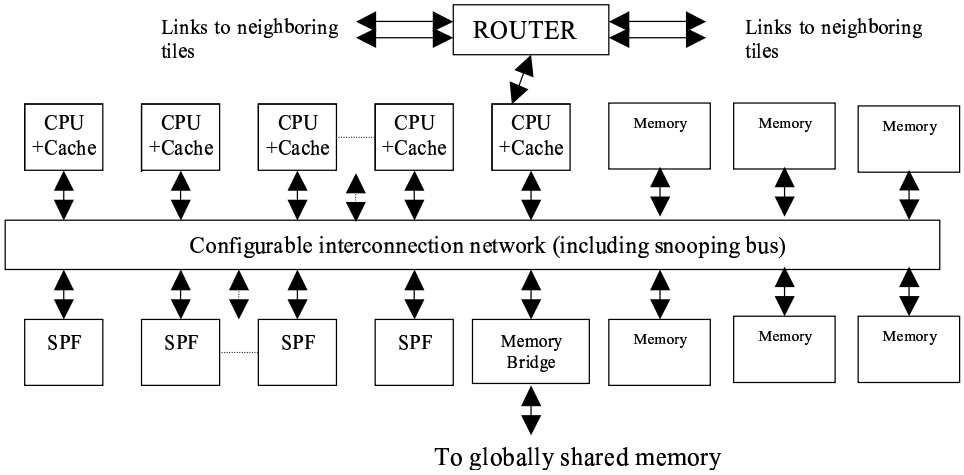

Fig. 1. SpaceCAKE Tile

3. The Verification Impasse

Verifying the functionality of the design is a key requirement. An effort to independently verify each individual module in the design using traditional simulation based verification will require Herculean effort and time. However, creating a top-level testbench for the verification of the architecture will lead to loss in the design visibility and hence the prolonged debug time. This impasse is resolved by the use of an assertion-based technique for the verification of the SpaceCAKE architecture. The assertion-based technique is implemented by embedding assertions into the design.

4. Implementation of Assertions

The different verification hot spots in the design were identified, and assertions were written to verify the hot spots. Verification hot spots denote design structures or features

of the design that are difficult to verify. Critical modules of the design that are highly depended upon by other modules are considered as hot spots since the total verification of the module is crucial for the operation of other modules. Simulation-based verification methodologies do not adequately verify hot spots because a typical verification hot spot processes too many combinations of events to be simulated exhaustively. A systematic verification methodology that employs exhaustive assertion-based dynamic and static tools is used to achieve verification signoff for the design.

Assertion Specification

Assertions expressed in Property Specification Language (PSL) [3] and Open Verification Library (OVL) [4] were embedded in the design. Though OVL provides ease of use because it comes as a predefined library, adapting the OVL to suit the needs of the design is complex. PSL provides the flexibility to express the functionality of the design explicitly and hence was used to a greater extent. PSL assertions are implemented in an external file, which is then explicitly bound to the modules of the design. This facilitates non-obtrusive use of the assertion-based technique with any verification set-up that is used.

Identifying hot spots

The hot spots identified can be classified into four categories.
1. Hot spots in complex design structures.
2. Hot spots in reusable protocol interface.
3. Hot spots in interface of different modules.
4. Hot spots in protocols and policies specific to the design.

Hot spots in complex design structures

SpaceCAKE consists of a number of design structures such as FIFOs, buffers, memories, arbiters, FSMs and pipelines. A few properties verified and the coverage information obtained in the above design structures are as follows.

FIFO

A queue is a FIFO structure. The FIFO structure has many corner cases that needs to be verified and is hence considered as a hot spot.

The properties verified for a FIFO include:
1. The FIFO does not accept data once full and does not refuse to accept data if not full.
2. Normal FIFO operation.
3. Data overwriting does not occur on FIFO overflow.

The coverage information obtained for a FIFO include:
1. Is FIFO size over-designed?
2. Is FIFO size under-designed?

BUFFER

The buffer design construct is mainly used for storing data temporarily before it is passed on to the next stage. Due to the high frequency of use of this construct, it is considered a hot spot.

The properties verified for a buffer include:
1. Simultaneous read and write to the same location of buffer is not performed.
2. Buffer indexing for reads and writes are consistent.
3. Width of data written into buffer is consistent with the width of the buffer.
4. Overwriting of data at a location, before a read on the location, is prohibited.

The coverage information obtained for a buffer include:
1. Are all the locations of the buffer read from and written into?
2. How many times is the buffer filled/emptied?
3. The numbers of read and write transactions.

MEMORY

Since the SpaceCAKE architecture is configurable, there are many memories that initialise the device operating conditions on boot and reflect the operation of the device over the course of time. Also, memories that facilitate table lookup operations are present.

These facts make memories an important hot spot to verify.

The properties verified for memories include:
1. The value of the memory on reset is as per specification.
2. Illegal/unknown values are not stored in memory.
3. Appropriate accessing of memory for read and write requests.
4. Configurable memories are initialized before certain time duration.

The coverage information obtained for memories include:
1. Reads and writes to all possible locations of the memory.
2. Number of read and write transactions.

ARBITER

An arbiter is an element that allows the bus to be shared by multiple devices. It is commonly used in all the SoC designs and provides many corner cases that should be verified.

The properties verified for an arbiter include:
1. The different arbitration schemes that can be implemented by the arbiter.
2. The request that has the highest priority is granted.
3. The grant to a request is within a certain number of cycles to the arrival of the request.

The coverage information obtained for an arbiter include:
1. Is every arbitration channel exercised?

2. How many simultaneous requests are made to the arbiter?

FSM

FSMs are design structures consisting of many legal paths. Each of these paths has to be verified to give the confidence of complete verification. This makes FSMs a hot spot for verification.

The properties verified for FSMs include:

1. Are the state transitions legal?
2. Has the FSM deadlocked or live-locked in any state?

The coverage information obtained for FSMs include:

1. Whether each state in the FSM has been exercised?
2. Whether every specified arc of the FSM is exercised?

PIPELINE

Pipelines form a crucial part of the present day designs as they help to increase throughput. Pipelines serve several concurrent actions that may be interdependent. This might lead to stalls in the pipeline. Therefore, the six-stage pipeline that forms a part of the SpaceCAKE design is a very crucial construct that has to be verified thoroughly.

The properties verified for pipelines include:

1. Every request is processed to completion.
2. Data corruption does not occur in the pipeline.
3. No live-locks/deadlocks in the pipeline.
4. Handshaking of signals between each stage.
5. Handling of pipeline stalls.

The coverage information obtained include:

1. Number and type of requests passed through the pipeline.
2. Number of pipeline stalls.

Hot spots in protocol interface

The data transfer protocol interfaces, used in the SpaceCAKE architecture, are the MTL [5], DTL [6], and AXI [7] protocols. Inclusion of assertions that completely defines the protocol at the interface ensures that any block that connects to the interface will be automatically checked for protocol compliance. Assertions to check the read/write operation of the protocol, address/data relationships, cycle latency and event scheduling time in the protocol (i.e. the time within which the interface should respond to a request) are used to define the MTL, DTL and AXI interfaces of the design. e Verification Components [8] that come along with these protocol checks were also used to verify these protocols.

Hot spots in inter-module interfaces

Interdependent modules are used in the design. Therefore, the verification of each module operating as a stand-alone unit is not sufficient. Use of interconnected modules can lead to deadlocks, if the modules are highly interdependent. Assertions that check the transmit/receive of valid data from each module, handshake behaviour modelling, the behaviour of communication between the two modules, and timeout conditions (indicate that a given module has not responded to a request within a particular duration) are implemented at the inter-module interfaces.

Hot spots in protocols and policies specific to the design

The DSM policy of the SpaceCAKE architecture is implemented as a lookup table that is programmed with the desired cache mode. This table lookup action is part of a pipeline and is therefore verified thoroughly. The two important design specific protocols are the cache coherency protocol and the snooping protocol. The cache coherency protocol implemented in the SpaceCAKE design is the MESI protocol [9]. The different cache modes possible are un-cached, bypass on miss, fetch on miss, and allocate on write. The snooping protocol allows four modes of read operations namely shared read, exclusive read, bus upgrade, bus invalid and two modes of write operation namely normal write and coherence write. Snooping ensures that all caches see and react to all bus events. Assertions to check whether each of these modes of operation is exercised by the design and whether the expected output is provided by the design once these different modes are activated are implemented.

The properties verified include:

1. Reads/writes in different cache modes.
2. Appropriate tag updates.
3. Cached transactions with snooping.

The coverage information obtained include:

1. The number of cache hits and misses.
2. Cache modes/snooping protocol coverage
3. The number of requests pertaining to the cache mode exercised.

5. Verifying assertions using static techniques

The static checking tool [12] was used to further verify the assertions specified. The assertions that passed the static check are completely verified for all possible input sequences. The staticverilog tool is used to initially compile the design and then statically check the design using design-partitioning techniques. The main limiting factor of this

style of verification is the time factor. Time consumed to verify the assertions statically is more than ten times the time taken to dynamically verify the assertion.

6. Dynamic verification using assertions

Dynamic verification involves generation of test stimulus to the design model. Generating test stimulus for the many individual modules of the SpaceCAKE design is a time consuming activity. Therefore, a top-level testbench is created. Though the utilization of a top-level testbench leads to loss of design visibility, the presence of embedded assertions alleviates this problem. The details of the SpaceCAKE test suite set up are given below.

Test suite set-up

The SpaceCAKE architecture is highly parameterised and configurable. Due to this fact the testbench set up for the design should be highly reusable. The top-level test environment set up for the architecture is as shown in Fig.2. Testbenches in C language are created to generate addresses and data corresponding to local and remote tiles. These C testbenches create Verilog tests that are applied to bus functional models of the processors. These are used to transfer the data to the SpaceCAKE interconnect network.

The tests issue read and write requests to the memory through a Bus Functional Model (BFM) of the processors. A BFM is a model that emulates the processor bus cycles. It does not execute the processor instruction set nor does it maintain the internal cache. The addresses generated by the tests are constrained to either fall in the local memory or in the remote memory that is part of the distributed shared memory space. The data generated by the tests is written at a particular address of the chip memory through the interconnect network and simultaneously into the reference memory. When a read request is generated to an address at which data was previously written into, the read data obtained by the read transaction is compared with the data stored in the reference memory. If the data matches then the write/read transactions have been performed correctly by the interconnect network.

These tests are self-checking and trigger an error if any discrepancy is found in the data transfer transactions.

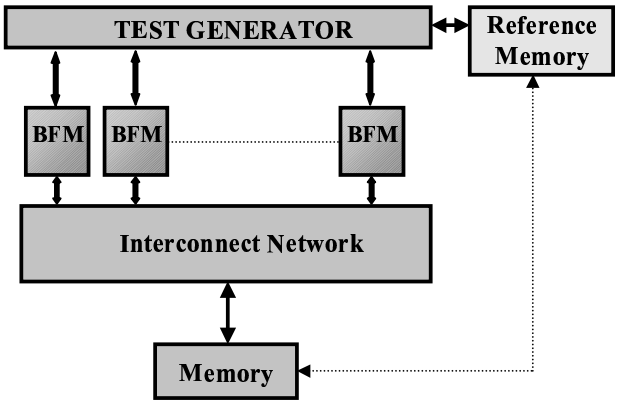

Fig. 2. Test set-up

The test set up was run on the design before and after the addition of assertions in the design. Tests that passed the design simulation run earlier, triggered assertions after the inclusion of assertions. This can potentially trigger the exposure of more bugs in the design. Also, the effort required to locate the bug once a test failed had considerably improved due to the high visibility provide by the assertions. The above set-up will check for acceptable write into memory and read from the memory operations for a single configuration of the interconnect network.

Activating different configurations

Typically the interconnect network can be configured in different ways as follows.
1. With remote or/and local memory access.
2. With and without debug functionality.
3. With/without different cache modes - fetch on miss, allocate on write, bypass on miss.
4. With/without snooping - exclusive read, bus upgrade, shared read, bus invalid.
5. With and without victimization/refill.
6. With DTL, MTL or AXI traffic.
7. With and without cache reservations.

Considering the high configurability of the interconnect network, a method of running the test suite using every configuration is needed to ensure that the interconnect functions correctly in all the different possible configurations. The manual selection of these configurations is time consuming and random selection may not yield meaningful values and sufficient confidence. Therefore, the different possible combinations were obtained by use of the Orthogonal Array Technique (OAT) tool [10][11]. The possible parameters

that can be configured along with the possible values that can be acquired by the parameter were provided as input to the tool. The tool generates the different possible configurations using the OAT method. The OAT technique was used so that the configurations obtained are well spread out and covers the state space uniformly without being redundant.

Regression environment

A regression environment was set up to run the test suite for different configurations. The environment also stores the waveforms, assertion firing reports, code and functional coverage reports pertaining to each test. This provides an automated way of running the different test suites. The code and functional coverage information reports obtained are merged to provide collective coverage information.

Coverage closure

Coverage closure is the process of finding areas of the HDL code not exercised by a set of tests so as to create additional tests to increase coverage by targeting holes, and determining a quantitative measure of coverage. The methodology depicted in Fig. 3 targeted coverage closures for the SpaceCAKE design.

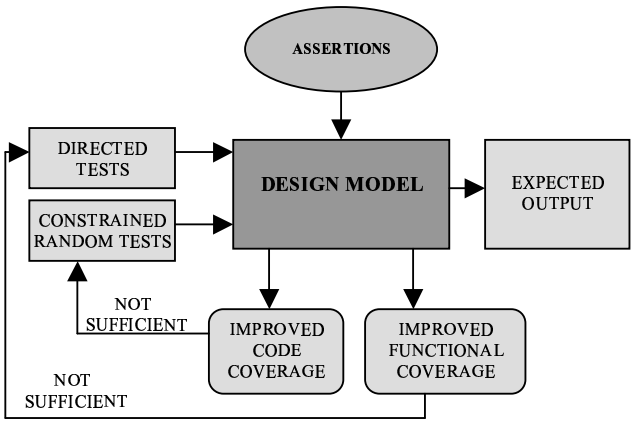

Fig. 3. Coverage closure

Once a collective code coverage report is available the areas of code that are not covered are manually checked for:

1. Non-accessibility of code with all configurations.
2. Presence of dead code.
3. Presence of bugs.

If the code coverage reports are satisfactory then the functional coverage reports are checked to ensure 100% functional coverage. Constrained random testing is used to increase the code coverage and directed tests are used to cover the caveats reported by the functional coverage tool.

7. Results

The top-level test suite in Fig. 2 was used to simulate the design before and after the addition of assertions. The bug detection rate increased since the same tests that had passed the design simulation run earlier triggered assertions, after the inclusion of assertions. Also, the task of locating the bug for the failed test considerably improved due to the high visibility provided by the assertions.

The following results were obtained after the test set-up was run with the embedded assertions.

- Bugs uncovered in the critical module of the design
  - Unassigned outputs.
  - Non-accessible code.
  - Illegal output values.
  - Errors in the lookup table.
- 50% reduction in debugging time.
- 10% of assertions completely verified using static checking. Static checking is unable to support part indexed select operator construct from Verilog2001 which is widely used in the design implementation. Also, restrict statements are not supported which leads to illegal input values being checked for. Due to the above limitations of the tool static checking could not be used extensively to verify the design.
- 62% of assertions were verified using dynamic verification.
- Code coverage of 95% for DSM functionality.

Observations: Assertion based technique

A comparative study of the different techniques adopted to verify the design, based on the results obtained, is given in this section. The use of assertion-based verification improved the productivity of the verification effort since it reduced the debug time by 50%. Increasing the density of the assertions in the design code will further reduce the debug time, however the simulation time overhead increases. This dilemma requires that the density of assertions be kept in check. The proficient placing of the assertions uniformly throughout the design implementation with increased concentration at the hot spots

facilitated the resolving of the dilemma. The reduction in debug time, with the use of assertion-based verification, is depicted in Fig. 4 and the overhead while using assertions is depicted in Fig. 5.

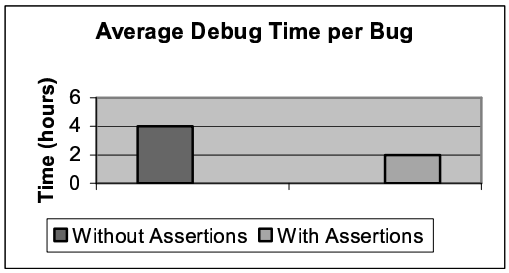

Fig. 4. Average Debug Time per Bug

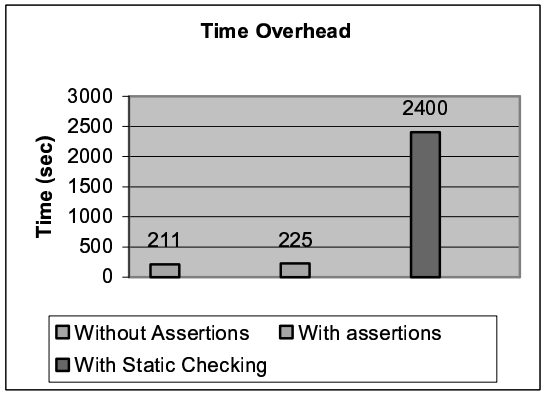

Fig. 5. Time Overhead

The addition of assertions in the design has caused only a 6% increase in the simulation time while using dynamic verification. This overhead can be overlooked considering the immense visibility provided to the design by the addition of assertions. The time overhead for static checking of assertions was more than ten times the time taken to verify the assertions dynamically. The advantage of using static checking though is to ensure the fact that the functionality has been verified completely.

Observations: Coverage metrics

Two different coverage metrics namely functional coverage and code coverage were obtained. The two coverage metrics complement each other. The graph in Fig. 6 depicts

the code and functional coverage obtained. The effort used to obtain functional coverage, is comparatively less while using assertions than using post-processing tests that perform sequence checking [13] etc. The graph emphasizes the fact that though 100% code coverage can be obtained for modules, the functional coverage reports give a true picture of the functionalities covered by the test suite. This is because all the paths in the design may not be covered with 100% code coverage. Only with a high code and functional coverage report can the design be released for tape-out.

Analysing the coverage metrics obtained indicates that for the design, presently, certain corner cases, multi-cycle paths and cross-correlated paths are yet to be verified.

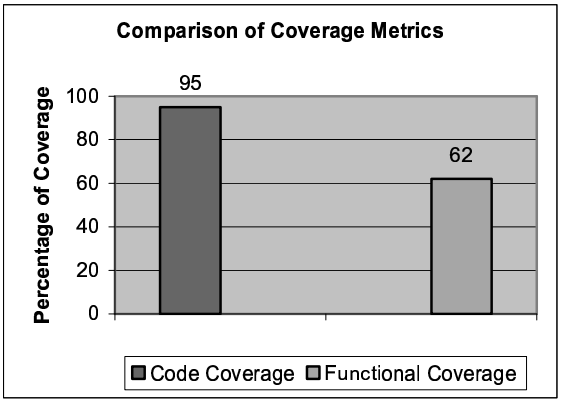

Fig. 6. Comparison of coverage metrics

Therefore, the verification effort needs to be directed towards verifying the corner cases and different paths of the design.

8. Conclusions

The key contribution provided by the use of the assertion-based technique, for verifying the SpaceCAKE architecture, is the remarkable increase in design visibility and therefore, a marked reduction in debug time. The assertions implemented can be used in concurrence with other verification techniques, like directed and pseudo random tests, to continuously run a background check and find unexpected bugs. Also, the use of the assertion-based functional coverage metric complemented with the code coverage metric provided a deeper insight into the areas of the design that needed further verification effort. This facilitated the verification effort to be directed towards verifying the key functionalities of the design.

To summarize, moving verification to the functional level, by focusing on protocol monitors, verification hot spots and critical coverage points provided a substantial return of investment. Assertion-based verification is an important methodology for the future generation complex chips. It is important for the EDA companies to make stronger tools around this methodology so that the verification effort is kept under control. As a methodology, it is a comprehensive start-to-finish approach, which not only integrates static, dynamic and coverage tools; it executes them in perfect coordination to accomplish a well-specified common goal.

9. References

1. Stravers. P, Hoogerbrugge. J, *Homogeneous Multiprocessing and the Future of Silicon Design Paradigms,* Proceedings of the IEEE International Symposium on VLSI technology, Systems and Applications, April 2001, pp. 184-187.
2. Stravers. P, *Homogeneous Multiprocessing for the Masses,* IEEE Workshop on Embedded Systems for Real-Time Multimedia, September 2004, pp. 3.
3. *Property Specification Language Version 1.1, Reference Manual,* Accellera, June 2004.
4. *Open Verification Library, Assertion Monitor Reference Manual,* Accellera, June 2003.
5. Philips Semiconductors, *CoReUse 3.2.1 Memory Transaction Level (MTL) Protocol Specifications.* Sept 2002.
6. Philips Semiconductors, *CoReUse 3.1.5 Device Transaction Level (DTL) Protocol Specifications.* Dec 2001.
7. *AMBA Advanced eXtensible Protocol v1.0 Specification.*
8. Cadence Specman Elite v4.3.4.
9. John Hennessy, David Patterson, *Computer Architecture and Organization,* Morgan Kaufmann Publishers, 3rd edition, 2003.
10. Sloane, Neil J. A. *A Library of Orthogonal Arrays.* Information Sciences Research Center, AT&T Shannon Labs. 9 Aug. 2001
11. *Jenny Tool.* http://burtleburtle.net/bob/math/jenny.html
12. *Incisive Static Assertion Checking Guide,* Product Version 5.1, October 2003.
13. M. Kantrowitz, Lisa M. Noack, I'm Done Simulating: Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha Microprocessor, Proceedings of the 33rd Design Automation Conference, June 1996, pp. 325-330.