Measuring the Effectiveness of Various Design Validation Approaches For PowerPC$^{\text{TM}}$ *Microprocessor Arrays

Li-C. Wang and Magdy S. Abadir

Somerset PowerPC Design Center
Motorola, Inc., Austin, Texas

Jing Zeng

Somerset PowerPC Design Center
IBM, Austin, Texas

Abstract

Design validation for embedded arrays remains as a challenging problem in today's microprocessor design environment. At Somerset, validation of array designs relies on both formal verification and vector simulation. Although several methods for array design validation have been proposed and had great success [[6], [9], [10], [13]], little evidence has been reported for the effectiveness of these methods with respect to the detection of design errors. In this paper, we propose a new way of measuring the effectiveness of different validation approaches based on automatic design error injection and simulation. This technique provides a systematic way for the evaluation of the quality of various validation approaches. Experimental results using different validation approaches on recent PowerPC microprocessor arrays will be reported.

1 Introduction

Among various design blocks in an advanced microprocessor, embedded arrays represent one of the most challenging parts for design validation. The arrays usually comprise large number of storage elements, together with complex timing and control logic that determine how they are used and updated. Arrays are custom designed so that their performance can be optimized to the greatest extent. In general, these arrays comprise more than 50% of the on-chip area and up to 80% of the total number of transistors [6].

In general, an array design is represented by three different views. They are: **the high-level (or RTL) view**, **the gate-level test view**, and **the transistor-level schematic view**. These views form the design data triangle. Each view on this triangle represents design data required for a particular set of tools and applications.

The RTL view is the most abstract, in the sense that it is the high-level specification of a design, where all functionality is described using high-level constructs. For custom design blocks such as arrays, transistor-level schematics are drawn according to the RTL specification. Usually, the gate-level views of these custom blocks are generated from the RTL model directly, which are used for test generation.

In a normal design flow, the RTL model for a module is first derived. Extensive functional simulation at the full chip level is carried out to verify the correctness of the RTL specification before the design is actually implemented. Then, various methods are applied to validate the correctness of these design views. These methods include:

- Simulation using manually generated test vectors.
- Simulation using tests generated by a commercial ATPG tool
- Formal verification based on Symbolic Trajectory Evaluation
- Simulation using tests generated based on formal verification results

Each of the methods has been reported with success [[6], [9], [10], [13]]. However, not much has been done in order to quantify the completeness of each approach with respect to design error detection. In this paper, we will present a systematic way to evaluate the quality of a validation method, based on logic error injection and detection. This approach provides an answer to the question of how complete a particular validation approach is. Experimental results based on applying various validation methods to recent PowerPC microprocessor arrays will be analyzed and discussed.

2 Overview of Various Validation Approaches

To validate the correctness of various design views, traditionally, vector simulation is used. This approach breaks down for arrays due to the exponentially large number of patterns required for exhaustive verification. Boolean equivalence checking is not completely suitable for arrays either because of the large number of storage elements and the complex timing scheme involved [8].

*IBM and PowerPC are trademarks of IBM Corporation in the United States, or other countries, or both

2.1 ATPG Based Test Vectors

One method that we have employed in the past with success to verify the equivalence between the gate level and the transistor level views of a design relied on the use of test vectors generated by ATPG tools and then simulating these vectors on the transistor level view [6].

Usually ATPG generated scan tests are used to cover all the logic surrounding the memory cells because core array elements are tested by Built-In Self Test (BIST) circuitry implementing certain marching algorithms [7]. In order for ATPG tools to understand the array logic, proper gate-level view is created. In practice, sequential ATPG tools are inadequate for handling arrays because of their large number of storage elements. Instead, arrays are modeled as combinational circuit with transparent primitives, such as predefined sequentially transparent latches and RAMs. Such a modeling approach is limited in the sense that the behavior of some designs can fall beyond the modeling capacity of these predefined primitives.

For high speed microprocessor design, arrays are designed with very complex timing scheme to optimize their performance. As a result, DFT engineers spend much of their time developing proper test view models for arrays using the available primitives. This process can be iterative and tedious. For the most complex array blocks, it is simply impossible to model them properly.

If an array can be modeled properly and a high fault coverage can be obtained, ATPG tests do have the advantage that they cover almost all structural sites in a circuit. Since design errors associated with a particular site whose stuck-at faults have been detected will likely be fortuitously detected [2], high fault coverage test set can be effective for detecting design errors.

2.2 STE Assertions

Formal verification with Symbolic Trajectory Evaluation (STE) [12] provides another alternative for validation. In STE, a set of *assertions* are created for an array design. Assertions represent a set of high level properties for which the design should satisfy *under normal operations*. Therefore, assertions can be thought as the golden model view of a design, which will be proved to be equivalent to all other views under STE. The advantage of this scenario is that not only the equivalence between two views can be verified but also the correctness of RTL model can be checked again independently. Therefore, if there is any unexpected error left in the RTL model after the extensive functional simulation, the error will be captured during the STE verification process.

STE [12] is a descendant of symbolic simulation [4]. At the core of symbolic simulation, Ordered Binary Decision Diagram [5] is used to efficiently manipulate boolean functions. In STE, a circuit is specified in terms of a set of properties in a restricted temporal logic form. Properties are expressed by so-called assertions which are of the form "**Antecedent $\Longrightarrow$ Consequent**" where both "Antecedent" and "Consequent" consist of a number of formulae. Formulae can be simple predicates such as "line A is 'a"' (line A holds the symbolic value 'a' at current time), or conjunctions of these simple predicates. Formulae can be applied with the next time operator in order to state the facts such as "line A remains 'a' in the next time step." It is also possible to apply *domain* restriction so that "line A remains 'a' when D is true" can be expressed. This simple logic in STE is sufficient for array verification purpose [[6] [9] [10]].

With STE, operations specified in the antecedent are symbolically simulated, and then conditions declared in the consequent are asserted. The assertion set represents a high level functional specification for a design. A Somerset in-house tool called *VerSys* built on top of *Voss* [11] is used to verify that all three views satisfy the same assertions.

There are two major limitations on the STE method. First, for large design, the OBDD size can easily blow up if too many symbols are used in a single assertion. Usually, selected non-critical symbolic values such as those used on the data path are replaced with constant values in order to reduce overhead. Second, the completeness of assertion set is not guaranteed. Since the STE verification is as complete as the corresponding assertions, it is possible that a particular design error will be missed due to the incompleteness of the assertion set.

Although STE provides the most comprehensive verification for arrays, simulation based methods are still required for the following reasons. Designer generated tests provide perhaps the cheapest and fastest way for detecting errors in the early design cycle. In addition, because of the limitations of the switch level simulation model [3], delays and other detailed electrical properties (crosstalk, power, noise) are not verified during STE, and hence detailed simulation is required for further design validation.

2.3 Assertion Based Test Vectors

Test generation based on STE assertion has been developed recently to combine the advantages of both ATPG method and STE approach. It has been shown that although "assertion tests" are generated without any structural information, they can achieve a higher stuck-at fault coverage than traditional ATPG tests due to the limitations of the ATPG tools on properly handling the arrays [13]. The basic idea is to replace the symbolic values used in the antecedent with a set of constant test vectors based on each condition specified in the consequent. The basic techniques include (details can be found in [13])

- replace a symbolic address comparison expression with *address marching* sequences
- replace a symbolic data comparison expression with *data marching* sequences
- replace stand-alone symbolic values with random vectors
- construct assertion decision tree and generate tests to cover all branches

- construct control signal decision tree in order to generate tests to cover abnormal functional space

Note that assertion tests are well-suited to validate design and to debug errors since they are based on the functional properties/operations of the arrays. They can also be used to detect defects and perform timing and noise analysis at the circuit level.

Since assertion tests do not exhaust all possible test space associated with an assertion, they are less complete than the assertion. On the other hand, assertion tests do explore the abnormal functional space which is *generally not considered by assertions, and hence provide additional check on the design.* One thing to note is that assertion tests use no structural information of the design, and thus may not provide a complete coverage on all sites as a 100% stuck-at fault test set is able to.

3 Design Error Model and Error Injection

None of the three approaches described earlier guarantees a complete validation. Stuck-at fault simulation can be used to provide some estimation of the quality of ATPG tests and assertion tests. This involves using a separate fault simulator other than the logic simulator such as Verilog. For STE verification, since the underline symbolic simulator does not have fault injection capability, it is difficult to perform fault simulation under the STE engine.

To evaluate the validation quality, we require a more systematic way which can be achieved with the existing logic simulator in use. We thus propose to use logic error design injection and simulation. This provides us a method to analyze the strength and weakness of various validation approaches, and hence quantifies how complete each approach is.

Several design error models were developed at the logic level in [6]. These models include: Extra inverter, Gate Substitution, Extra wire, Extra gate, Missing gate, Wrong wire, etc. For initial experiments, we will consider only the simple types of error described above. However, the framework can be extended to cover other types of design error models easily.

Since simulating all possible design errors is usually too expensive, a set of $n-1$ design errors are inject randomly, where n is usually linearly proportional to the number of gates in a design. To inject $n-1$ random errors, we modify the design in the following way. $\log(n)$ additional primary inputs are first added to the design. These additional inputs are then decoded to select $n-1$ errors, plus one case where no error is injected. Error injection is essentially controlled by a multiplexer, where either the good design or a faulty design is selected locally. Figure 1 shows the modification for the extra inverter error model.

Thus with control input bus of width $\log(n)$, there are total of n possible input combinations where $n-1$ can be used to select design errors. The modified design is then fed into a logic simulator. Expected results from the original design are compared and if there is a mismatch, a detection is reported. Note that parallel error simulation (similar to parallel fault simulation) is not implemented here due to the sequential nature of array designs, and because an existing logic simulator is used, only one error can be simulated at a time.

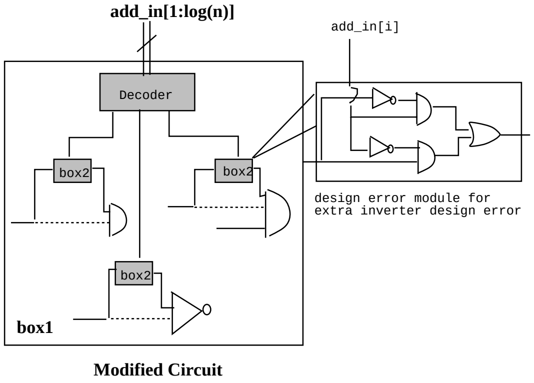

Figure 1: Error Injection of an Extra Inverter

4 Results

For the initial experiment, an eight-way set associative tag array design is selected. The control logic surrounding the memory core consists of around 5500 gates. As described earlier, memory core and the address decoder are tested by Built-In Self Test (BIST) circuitry implementing certain marching algorithms [7]. ATPG tests are used to cover only the surrounding logic. Since memory core is constructed in a regular way and is usually not a major concern for validation purpose, design errors are injected only in the surrounding logic as well.

Two sets of 1023 design errors were randomly injected for the two experiments denoted as EXP I and EXP II. This creates 10 additional primary inputs to the design. Design error coverages are obtained for ATPG tests and for the assertion tests. Table 1 shows the stuck at fault coverage comparison between the two test sets, given that the fault coverage by assertion tests has been normalized to 1.

Tables 2 and 3 present the comparison of design error detection by the two test sets. For a given normalized fault coverage figure, assertion tests are generally better than the ATPG tests with respect to design error detection, except for normalized FC = 0.6 where ATPG tests temporarily outperform assertion tests.

At the end, assertion tests achieve higher error coverage. Such results are expected given the higher fault coverage by the assertion tests. As it has been established in [2], stuck-at faults and logic design errors considered here are highly correlated. Hence, although it is not necessarily true, in general higher stuck-at fault coverage does imply a higher design error coverage.

Note that undetected design errors could be redundant. Further research efforts are required to explore the nature of those errors. An interesting result to note was that in both Table 2 and 3 *all design errors detected by the ATPG tests were also detected by the assertion tests.* However, this is not true for the stuck-at fault results shown in Table 1.

Normalized Results	ATPG	Assertion Tests
SAF Coverage	0.8	1.0

Table 1: Results of Normalized SAF Coverages

Errors Detected	ATPG Tests	Assertion Tests
at FC 0.5	578	642
at FC 0.6	672	643
at FC 0.7	717	776
at FC 0.8	728	917
Final #	728	965

Table 2: Results of Design Error Detection at Different Normalized Fault Coverages (FC) - EXP I

Errors Detected	ATPG Tests	Assertion Tests
at FC 0.5	618	644
at FC 0.6	760	658
at FC 0.7	784	816
at FC 0.8	809	891
Final #	809	932

Table 3: Results of Design Error Detection at Different Normalized Fault Coverages (FC) - EXP II

To better understand the correlation between the ATPG tests and assertion tests, Figure 3 in the Appendix shows the normalized fault coverage curves (with respect to the number of tests applied) for assertion tests and ATPG tests. Figure 4 in the Appendix shows the design error detection curves for experiment I. The curves for experiment II are similar and hence, are omitted.

It can be seen that ATPG tests are very effective for the first 300 vectors, and then become relatively ineffective for the remaining vectors. This is true for detecting both the stuck-at faults and the design errors.

To understand the behavior of the assertion tests, it is necessary to explain how those tests are generated first. In Figure 2, assertion tests are divided into six different blocks. The first block is for setting the initial content of some of the array cells. Then, the next three blocks contain both address marching and data marching sequences which are generated for the two major assertions, Tag Store (write) and Tag Load (read). The fifth block is for other assertions that treat the Tag as a static cache array. No marching is used here. Instead, random selected addresses are used. The last block contains tests from the control signal trees for all assertions in order to cover the abnormal functional space.

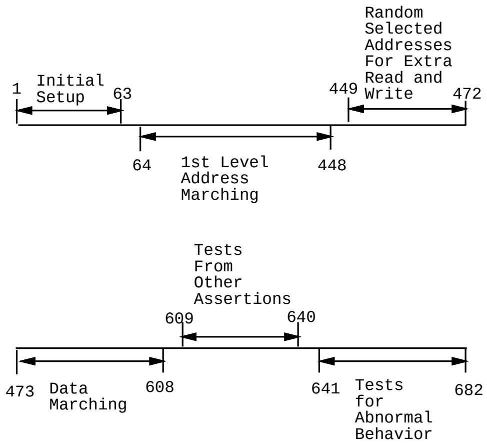
1
Initial Setup
63
Random Selected Addresses For Extra Read and Write
449
472
64
1st Level Address Marching
448
Tests From Other Assertions
609
640
473
Data Marching
608
641
Tests for Abnormal Behavior
682

Figure 2: Illustration of Assertion Vectors

It can be seen that address marching tests are less effective with respect to both stuck-at fault and design error detection. This is because no faults or errors were injected in the decoder circuitry, and hence, address marching tests which primary target on the decoder and memory cell faults are not of much use. On the other hand, all other blocks of tests are very effective and necessary for detecting both faults and design errors.

It is interesting to note that tests for abnormal functional space do detect unique faults and design errors, and hence cannot be ignored.

Table 4 demonstrates the design error detection results by STE assertions. It is interesting to note that these symbolic assertions do not outperform the assertion tests although the later are generated based upon the former. This is because *the assertions considered here are constructed to verify the normal functional space only while assertion tests include vectors specifically targeting on those abnormal space.* Hence, a design error that changes the behavior of a circuit in the abnormal functional space (such as "don't care" area) will usually not be considered as a true error by assertion verification. Note that assertions for detecting errors in the abnormal functional space can also be achieved.

Assertions	Store	Load	Others	Total
(EXP I)	282	758	373	964
(EXP II)	256	709	340	929

Table 4: Results of Design Error Detection by Symbolic Assertions

5 Conclusion

In this paper, three different validation approaches are investigated for their effectiveness with respect to design error detection. The initial results demonstrate that design error injection is a feasible approach to evaluate the quality of various validation approaches. For all experiments, no additional special simulator is required. Logic simulation is used to measure the effectiveness of ATPG tests and assertion tests while the STE symbolic simulation engine is used for assertions. Current results indicate that assertion tests are the most comprehensive approach among the three. However, STE assertions can be improved to detect more design errors if individual assertion is included for each abnormal operation in the assertion control signal tree.

References

[1] Magdy S. Abadir, and H. K. Reghbati, *Testing of random access memories*, ACM Computing Surveys, September 1983.

[2] Magdy S. Abadir, Jack Ferguson and Tomas E. Kirkland, *Logic Design Verification via Test Generation*, IEEE Transactions on Computer-Aided Design, Vol.7, No.1, January 1988.

[3] Randal E. Bryant, *A switch-level model and simulator for MOS digital systems*, IEEE Transactions on Computers, Vol C-33, No. 2, February 1984, pp. 160-177.

[4] Randal E. Bryant, *Symbolic simulation — techniques and applications*, 27th Design Automation Conference, 1990.

[5] Randal E. Bryant, *Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams*, ACM Computing Surveys, Vol. 24, No. 3, Sep. 1992.

[6] Neeta Ganguly, Magdy S. Abadir, and Manish Pandey, *PowerPC Array Verification Methodology Using Formal Verification Techniques*, International Test Conference, Washington DC., 1996. pp. 857-864.

[7] C. Hunter, J. Slaton, J. Eno, R. Jessani, C. Dietz, *The PowerPC603(tm) Microprocessor: An Array Built-In Self Test Mechanism*, International Test Conference, 1994, pp. 388-394.

[8] Charles H. Malley and M. Dieudonne, *Logic Verification Methodology for PowerPC Microprocessors*, 32nd Design Automation Conference, 1995, pp. 234-240.

[9] Manish Pandey, Richard Raimi, Derek L. Beatty, and Randal E. Bryant, *Formal verification of PowerPC$^{\mathrm{TM}}$ arrays using symbolic trajectory evaluation*, 33rd Design Automation Conference, Las Vegas, NV., 1996.

[10] Manish Pandey, Richard Raimi, Randal E. Bryant, and Magdy S. Abadir *Formal verification of Content Addressable Memories Using Symbolic Trajectory Evaluation*, to appear in 34rd Design Automation Conference, 1997.

[11] C,J.H. Seger *Voss — a formal hardware verification system: user's guide*, Technical Report 93-45, Department of Computer Science, University of British Columbia, 1993.

[12] C,J.H. Seger and Randal E. Bryant, *Formal verification by symbolic evaluation of partially-ordered trajectories*, Formal Methods in System Design 6, 1995, pp. 147-189.

[13] Li-Chung Wang, and Magdy S. Abadir, *A New Validation Methodology Combining Test and Formal Verification for PowerPC^{TM} Microprocessor Arrays*, to appear in International Test Conference, 1997.

Appendix

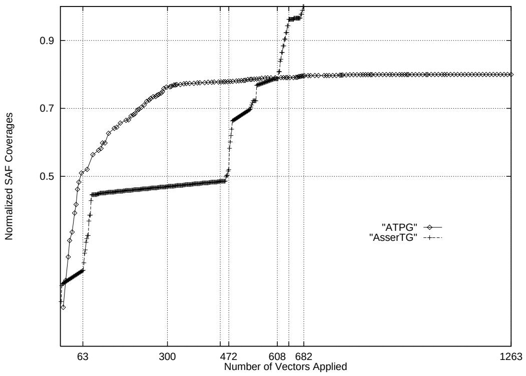

Figure 3: Normalized Stuck-at Fault Coverage Curves: ATPG Vs. Assertion Vectors

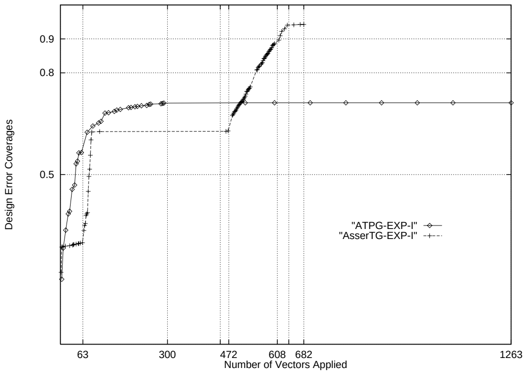

Figure 4: Design Error Coverage Curves (EXP I): ATPG Vs. Assertion Vectors