JOURNAL OF ELECTRONIC TESTING: Theory and Applications, , 1–16 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Test Generation Based On High Level Assertion Specification For PowerPC™* Microprocessor Embedded Arrays

LI-C. WANG
Somerset PowerPC Design Center, Motorola Inc.,
6200 Bridgepoint Parkway, Bldg 4, Austin, Texas 78730
lwang@ibmoto.com

MAGDY S. ABADIR
Somerset PowerPC Design Center, Motorola Inc.,
6200 Bridgepoint Parkway, Bldg 4, Austin, Texas 78730
abadir@ibmoto.com

;

Editor: Niraj Jha

Abstract. Test and validation of embedded array blocks remains a major challenge in today's microprocessor design environment. The difficulty comes from twofold, the sizes of the arrays and the complexity of their timing and control. This paper describes a novel test generation methodology for test and validation of microprocessor embedded arrays. Unlike traditional ATPG methods, our test generation method is based upon the high level assertion specification which is originally used for the purpose of formal verification. The superority of these assertion tests over the traditional ATPG tests will be discussed and shown through various experiments on recent PowerPC microprocessor designs.

Keywords: High Level Test Generation, Assertion Test Generation, Design Validation, Logic Verification, Symbolic Trajectory Evaluation

1. Introduction

Embedded arrays (such as cache, cache tag, look-up tables, register files, etc.) are complex sequential logic blocks which represent one of the major challenges in modern microprocessor design environment. The challenge comes from two ways: the size of these arrays and the complexity of their designs. For example, typical arrays in **PowerPC** microprocessors may contain from 200 thousand to 2 million transistors, which may consist of tens of thousands or hundreds of thousands storage elements. These storage elements are usually modeled in terms of predefined primitives (such as transparent RAMs) which can be understood by the Automatic Test Pattern Generation (ATPG) tools. There are two disadvantages with this approach. First, usually it is a tedious and painful process to model complex array behavior in terms of a set of predefined primitives. More seriously, sometimes it is difficult to model arrays perfectly for traditional ATPG tools to capture the complex timing and control. As a result, fault coverage

*IBM, PowerPC, and PowerPC603e are trademarks of IBM Corporation in the United States, or other countries, or both

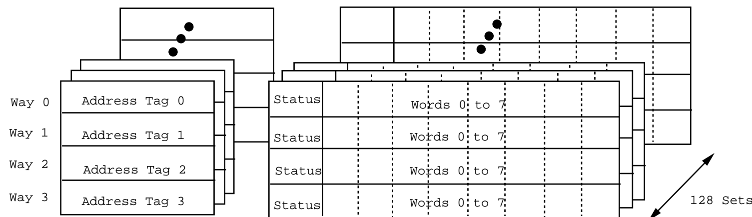

Fig. 1. Instruction Cache Organization

loss becomes inevitable and extensive Design-For-Testability (DFT) works are usually required.

In today's RISC microprocessors, arrays may contain up to 80% of the transistors in use. Hence, their validation and test becomes increasingly important. While traditional ATPG approaches fail to provide a good and economical solution to the problem, in this paper we will present a novel test generation methodology for arrays (preliminary results were also presented in [14]). Unlike traditional ATPG approaches, our method is based upon the high level array assertion specification. High level assertion specification is originally for the purpose of verification. To verify the correctness of an array design, at first a set of *assertions* are created. Assertions are a set of antecedent and consequent pairs which express the functionality of an array design in the temporal logic form [10]. The actual implementation is formally verified to be equivalent to the specification using an in-house formal verification tool based on Symbolic Trajectory Evaluation (STE) [13]. Then, the new test generation method utilizes the existing assertion specification which interprets a design in the form very different from the common Register-Transfer-Level (RTL) view. Without going through the lower gate level view, the new approach avoids the extensive DFT works.

The paper is organized as follows. In Section 2, we review test and verification methodology and identify the problems of test generation for arrays. Sections 3 explains how assertions are applied to verify arrays using STE. In Section 4, techniques for generating assertion tests are described in detail. To demonstrate the superiority of these tests, in Section 5 assertion tests are compared against ATPG tests based upon the single stuck-at fault model at both gate and transistor level, as well as based upon the logic design errors. Encouraging results are obtained on array designs from recent PowerPC microprocessors.

2. Background

Embedded arrays in PowerPC microprocessors are varied and complex. They can be as simple as register blocks like general-purpose registers (GPRs), floating-point register (FPRs), and segment registers (SRs), or more complex like the block address translation unit (BAT), table look-up buffer (TLB), and so on. Large arrays include instruction cache (ICACHE), data cache (DCACHE), and their corresponding tags (ITAG and DTAG).

As an example, consider the instruction cache in PowerPC 603e™ microprocessor [16]. The structure is illustrated in Figure 1. The cache is configured into four ways each with 128 sets. A set contains 8 contiguous words from memory. Several status bits are used to implement the LRU (least recently used) algorithm for cache replacement. There are usually multiple read and write ports to simultaneously access the cache. Moreover, there is control logic to decide how read/write conflicts should be resolved.

The arrays comprise storage elements, together with complex timing and control logic that determine how they are used and updated. These arrays in our recent processors comprise more than 50% of the on-chip area, and up to 80% of the total transistors in use.

In a regular design flow, an array design is represented by three views. They are:

- the **high-level** (or RTL) view
- the **gate-level** view
- the **transistor-level schematic** view.

These three views form the design data triangle as shown in Figure 2. Each view on this triangle

represents design data required for a particular set of tools and applications.

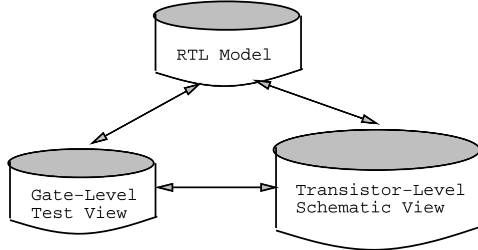

Fig. 2. Design Representations

The RTL view is the most abstract, in the sense that it is a high-level specification of the design, where all functionality is described using high-level constructs. Usually, the gate-level views are generated from the RTL models directly, which are used for the test generation purpose.

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 view before the design is actually implemented. Normally, the RTL model is passed to the synthesis tools to generate a schematic view. For some blocks such as embedded arrays, since performance is crucial, they are custom designed so that timing can be optimized as much as possible. This ends up with very complex control so that many operations can be squeezed into very few short clock cycles.

2.1. Logic Verification

The goal of logic verification is to verify the equivalence between the design specification (the RTL view) and the design implementation (the schematic view). In the design data triangle in Figure 2, all three views should be equivalent. Traditionally, vector simulation is used for logic verification. Due to the exponentially large number of patterns required for exhaustive verification, vector simulation is always questionable regarding its completeness. Traditional boolean equivalence checking is not completely suitable for arrays either because of the complex timing scheme involved and mismatch among latches in different views [9]. One method that we have employed in the past to verify the equivalence between the gate level and the transistor level views of a design relies upon the use of ATPG test vectors generated on the gate level view and simulation of these vectors on the transistor level schematic view [7].

2.2. Introduction of the Verification View

To properly verify the design views, formal verification with Symbolic Trajectory Evaluation (STE) [13] provides a feasible alternative [7]. In this new verification methodology, a set of *assertions* are created for a design (details will be described in the next section). Assertions represent a complete set of high level properties for which the design should satisfy under normal operations. Assertions can be thought as the *golden forth view* of a design, which can be proved to be equivalent to all others under STE. Hence, while the RTL view is devised for the purposes of functional simulation and logic synthesis, assertion verification view exists for the reason of formal verification.

The advantage of this scenario is that not only the equivalence between any two views can be verified but also the correctness of RTL model can be checked again independently. Therefore, if there is any unexpected functional/logic error left in the RTL model after the extensive functional simulation, the error will be captured during the STE verification process.

Although STE provides the most comprehensive verification for arrays, simulation based methods are still required for the following reasons. Designer generated tests for embedded arrays provide perhaps a quick way for detecting errors in the early design cycle. In addition, because of the limitations of the switch level simulation model [4] used in STE, delays and other detailed electrical properties (crosstalk, power, noise) are not verified, and hence more detailed simulation is required for further validation of various electrical properties.

2.3. Test and Test Generation

Testing for arrays is achieved in two ways. For the core array elements such as memory cells, they are tested by Built-In Self Test (BIST) circuitry implementing certain marching algorithms [8]. ATPG generated scan tests are used to cover

all other control logic surrounding the memory cells. At our design center, all array modules are fully scannable. To enable the ATPG tools to understand the array logic, proper gate-level view is created. Usually, the gate-level view is synthesized directly from the RTL model using an internal tool so that test generation does not wait until the actual design implementation is finished. This allows us to obtain test patterns as early as possible in the overall design flow.

As described in the Introduction, test generation on arrays represents a major challenge in the overall test and validation flow.

Due to their large number of storage elements, arrays are usually modeled with a set of predefined higher level primitives, such as RAM and latches. Once an array can be modeled in this way, ATPG tools understand how to propagate an input signal through a primitive to its output. For example, a sequence of predetermined control and clock signals is able to propagate an input of the RAM to its output. Such sequence comprises so-called *test procedure*. With test procedures, ATPG tools can simplify its view on sequential elements. This scenario works fine for arrays if they can be perfectly modeled by the given primitives.

For high speed microprocessor design, arrays are crucial for performance. Therefore, they are designed with very complex timing scheme to optimize their operation speed. As a result, DFT engineers spend tremendous amount of efforts to develop proper gate level test view models for arrays using the available primitives so that they can be fully understood by the ATPG tools. Without the DFT work, fault coverage loss is inevitable. For the most complex array blocks, sometimes, it is simply impossible to model them properly.

Due to the limitations of ATPG tools and the process of DFT, ATPG generated tests based on the gate level test view have to be simulated at the transistor level in order to verify their correctness. Typically, some of these vectors will fail the simulation. Then, the process of "modifying the test model → generating tests → simulating them at the transistor level" has to be iterated until all vectors pass. The process can be very tedious.

In summary, test generation for complex array blocks poses a major challenge in test and validation methodology.

- Gate-level test views are created and modified to be used solely for the ATPG purpose. This extra view imposes additional burden on the verification.
- Properly modeling an array with ATPG understandable primitives is a tedious process, involving creation of extra logic and/or changes of the primitives. Such DFT works are necessary to obtain high fault coverages on arrays.
- Regardless of how hard we may try, for some array designs, traditional ATPG tools are simply too limited to handle them, resulting in unsatisfactory fault coverages.

2.4. Assertion Test Generation

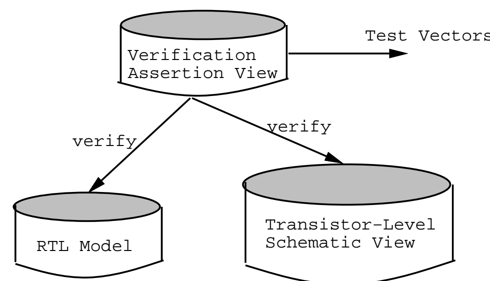

Fig. 3. New Design Representations

Test generation from the high level assertion specification provides an ideal solution to the problem of both test and verification. If tests can be derived directly from assertions, DFT on the gate level test view is no longer required and hence, eliminate the verification needs associated with it. This also avoids the troubles of going through the ATPG tools. Hence, instead of creating complex gate-level views for arrays so that they can be tested through ATPG tools, the new flow utilizes the high level assertion specification which already exists for the purpose of STE verification. Figure 3 shows the resulting different design data perspective from that in Figure 2. In the Figure, both the RTL view and schematic view are verified to satisfy the verification assertion view while test vectors are derived directly from the assertions.

Assertion tests are more like functional tests for the individual array block. They can be used to debug a design after STE verification discovers

that there is a discrepancy. From this perspective, assertion tests are much more useful than ATPG tests because it is usually harder to capture the functional intention of a sequence of ATPG patterns. For large designs where STE formal verification becomes inefficient due to the large amount of memory required, assertion tests provide an inexpensive and effective alternative for verification. Finally, assertion tests will be used to detect manufacturing defects and hence, replace the ATPG tests.

In the following, we will first describe how assertion specification is constructed and also introduce STE formal verification approach for arrays. Then, test generation from assertions is presented. Although we have described many advantages with the assertion test generation, it remains to be shown that assertion tests are superior to ATPG tests from the quality perspective. In Section 5, comparison results will be reported.

3. Assertion Specification and STE Formal Verification

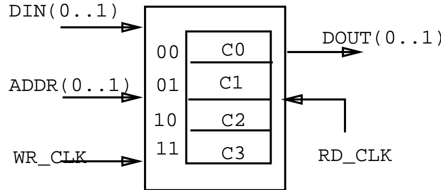

Fig. 4. A simple array example

As described before, assertion specification for array exsits for the purpose of STE formal verification. STE [13] is a formal verification technique which is a descendant of symbolic simulation [5]. At the core of symbolic simulation, Ordered Binary Decision Diagram [6] 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 the array verification purpose [7] [10] [11].

As an example, consider a simple array with only four memory cells C0, C1,C2, and C4 (2 bits each), one write port with a write clock, and one read port with a read clock. The array structure is shown in Figure 4. There are two operations defined for this array: read and write. For read, ADDR will be supplied with the address plus RD_CLK being set to high and WR_CLK being set to low. DIN is ignored for read. For write, DIN holds the data and ADDR points to the address with RD_CLK being set to low and WR_CLK being set to high. If both clocks are low, then there is no action on this array. It is illegal for both clocks to be high in the sense that the surrounding logic will guarantee this never happens. To specify the write operation, a "write" assertion can be very much like the following.

Initialization:	$Array[A] = D$ at time 0
	where symbolic address $A = a_0a_1$
	and symbolic data $D = d_0d_1$
Antecedent:	(Set controls to write)
	DIN$= C$ and ADDR$= B$
	during time period $(0, T]$
	where $C = c_0c_1$ and $B = b_0b_1$
$\Longrightarrow$ (lead to)	
Consequent:	at time T,
	if $(A = B)$ then $Array[A] = C$
	else $Array[A] = D$

Separate procedures will be done to specify the details of how to initialize the array and how to write data into it. For instance, for write operation, it may involve keeping RD_CLK to be low for all time, and setting WR_CLK to be high from, say time $T/4$ to $3T/4$ while keeping it to be low for the remaining periods. The time stamps such as "$T/4$" "$3T/4$" and "T" can be arbitrarily selected as long as it guarantees that in the symbolic simulation all signal changes stablize before the next input changes occur. In other words, the relative order of these time stamps are important, not their absolute values. The power of such an

assertion lies in the use of *symbolic indexing*, such as A and B. With symbolic address A, a simple reference in the consequent such as "$Array[A]$" actually covers all cells in the array.

With STE, operations specified in the antecedent (including the initialization which is treated as part of the antecedent) are symbolically simulated, and then conditions declared in the consequent are asserted. For example, at time T, symbolic simulation obtains the following boolean expressions if the array is correctly implemented (where X is the ternary constant defined in the simulation [4]):

Table 1. Results of Symbolic Simulation

Let $(A = B) := (a_0a_1b_0b_1 + a'_0a_1b'_0b_1 + a_0a'_1b_0b'_1 + a'_0a'_1b'_0b'_1)$		
Let $(A \neq B) := (a_0a_1b_0b_1 + a'_0a_1b'_0b_1 + a_0a'_1b_0b'_1 + a'_0a'_1b'_0b'_1)'$		
After Symbolic Simulation:		
bit 0 of cell C0	=	$(A = B)[b'_0b'_1c_0 + (b_0b'_1 + b'_0b_1 + b_0b_1)X] + (A \neq B)[a'_0a'_1d_0 + (a_0a'_1 + a'_0a_1 + a_0a_1)X]$
bit 1 of cell C0	=	$(A = B)[b'_0b'_1c_1 + (b_0b'_1 + b'_0b_1 + b_0b_1)X] + (A \neq B)[a'_0a'_1d_1 + (a_0a'_1 + a'_0a_1 + a_0a_1)X]$
...		...
bit 0 of cell C3	=	$(A = B)[b_0b_1c_0 + (b_0b'_1 + b'_0b_1 + b'_0b'_1)X] + (A \neq B)[a_0a_1d_0 + (a_0a'_1 + a'_0a_1 + a'_0a'_1)X]$
bit 1 of cell C3	=	$(A = B)[b_0b_1c_1 + (b_0b'_1 + b'_0b_1 + b'_0b'_1)X] + (A \neq B)[a_0a_1d_1 + (a_0a'_1 + a'_0a_1 + a'_0a'_1)X]$

These symbolic simulation results are then compared against the conditions specified in the consequent. In this case, they match and the design passes the assertion. Otherwise, the process of STE fails. An in-house tool called *VerSys* built on top of *Voss* [12] is used to verify that RTL and schematic views satisfy the same assertion specification.

In the following, we will describe the specifications of several classic designs in an array. In the next section, we will explain how to generate tests from these array specifications.

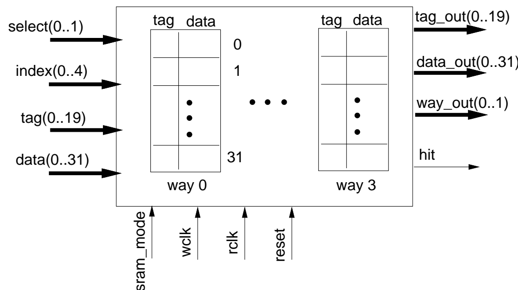

Fig. 5. A simple TAG example

3.1. Static RAM

First example is a simple tag array design as shown in Figure 5. This array has been simplified from an actual tag design but still captures several important features in common tag arrays. In this design, core memory can be divided into 4 blocks where $select(0..1)$ selects which way to be operated on. For each memory block, address is given through $index(0..4)$ for selecting one out of the 32 locations. There are three major operations defined in this array: static write, static read, and tag load.

static write For a static write, $sram_mode$ is set to high. $wclk$ is high and $rclk$ is low. $reset$ is kept low. A 20-bit tag is placed at $tag(0..19)$ while the corresponding 32-bit data are at $data(0..31)$.

static read Similarly, $sram_mode$ is set to high. $rclk$ is high and $wclk$ is low. $reset$ is kept low. Tag is read out at $tag_out(0..19)$ while the corresponding data are read out at $data_out(0..31)$.

tag load The incoming tag is placed at $tag(0..19)$. $sram_mode$ is set to low. $rclk$ is high and $wclk$ is low. $reset$ is kept low. The tag is compared to the tags stored at location $index(0..4)$ of all 4 memory blocks. If there is a match, then the corresponding tag and data are read out. $way_out(0..1)$ will point to the matched block and hit becomes high. Otherwise, hit is low.

To specify the static write operation, the assertion can be the one shown in Table 2 (time stamps are omitted for simplicity). First, all array blocks are initialized with potentially different symbolic values using the same symbolic index A. Then,

a symbolic selection vector S is given so that at one write, all four ways can be checked simultaneously. The consequent for each of the four ways is similar to the "simple write" assertion described earlier except that in addition we should specify that all other array blocks are not affected by the write.

Table 2. Assertion for Static Write

Initialization:	$Array_i[A] = T_i D_i$,
	$\forall i = 0, 1, 2, 3$
Antecedent:	(set controls to write)
	tag= T, data= D,
	index= I, select= S
$\Longrightarrow$ (lead to)	
Consequent:	for $k = 0, 1, 2, 3$
	if $(S = k)$
	if $(A = I)$ $Array_k[A] = TD$
	else $Array_k[A] = T_k D_k$
	$\forall j \neq k, Array_j[A] = T_j D_j$

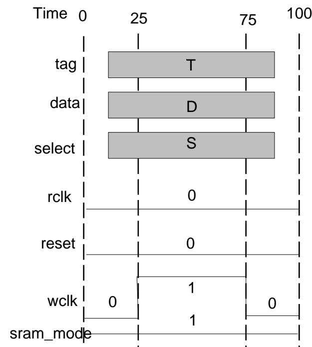

Fig. 6. Timing and control settings for static write

With symbolic indexing and symbolic way selection, the assertion for static read can be constructed similarly. Note that separate procedure is needed to set the control signals to achieve a read or a write in the antecedent. For example, for a write operation, the procedure may involve setting $rclk$ to be low, $sram_mode$ to be high, and $reset$ to be low from time 0 to 100, plus setting $wclk$ to be high from time 25 to 75. This timing and control settings are illustrated in Figure 6.

3.2. Tag Load

The assertion for tag load is described in Table 3. Note that in the consequence part, it guarantees that there is *at most* one hit. This is achieved using the *domain* operator "**when**".

Table 3. Assertion for Tag Load

Initialization:	$Array_i[A] = T_i D_i$,
	$\forall i = 0, 1, 2, 3$
Antecedent:	(set controls for tag load)
	tag=T, index= A
$\Longrightarrow$ (lead to)	
Consequent:	**if** $(T_k = T)$ for $k = 0, 1, 2, 3$
	$data_out() = D_k$, $tag_out() = T_k$,
	$way_out() = 00$, $hit = 1$
	when $(T_k \neq T_j) \forall j \neq k$
	if $\forall k \in \{0, 1, 2, 3\} (T \neq T_k)$
	$hit = 0$

3.3. Pseudo LRU

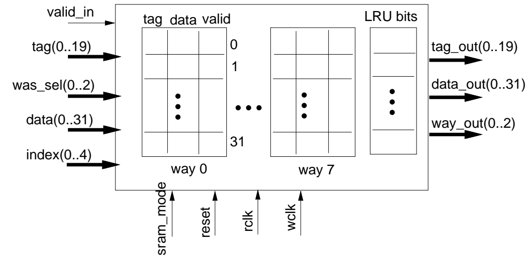

Fig. 7. An LRU example

Figure 7 shows a design similar to Figure 5 except that this time, a pseudo LRU control is included. A new operation called *replace* can be defined with *sram_mode* being set to low. *Replace* is a write operation where the *Least Recently Used* block defined by the LRU bits is replaced. Hence, in this case *way_sel* is ignored. Also, during a read or write, the LRU bits should be modified to reflect the facts the corresponding block entry is recently being accessed.

The pseudo LRU control algorithm (which may not implement a true LRU) usually involves two sets of rules. In the first set, the rules control which way to be replaced based on the LRU bit values. For example, if the LRU bits are "00x0xxx" then it is way 0 to be replaced. The second rules control how to modify the LRU bits after an entry is accessed. For example, "11x1xxx" are ORed with the exist bits if way 0 is being accessed. We will use R_i to denote the replacing rule and M_i to denote the modification

rule for way i. Note that the replacing rules are mutually exclusive.

In the LRU, a new array "valid" is added for each block. The additional valid bits also control the replacement. When a replacement is needed, the control logic will scan the valid bit from way 0 to way 7 and replace the first way whose valid bit is zero. Only if all valid bits are 1 will the LRU algorithm be involved. Also, for the "tag load" operation, only those tags whose corresponding valid bits are 1 will be used for comparison.

Table 4. Assertion for LRU

Initialization:	$\forall k = 0 \ldots 7$	
	$Array_k[A] = T_k D_k v_k$	
	$LRU[A] = L$	
Antecedent:	(set symbolic controls for replace)	
	tag= T, index= A, $data = D$	
$\Longrightarrow$ (load to)		
Consequent:	for $k = 0 \ldots 7$	
	if	$(v_k = 0)$
		$Array_k[A] = TD1$
		$LRU[A]$ is modified by rule M_k
		other blocks are not affected
		Stop the consequent check here.
	for $k = 0 \ldots 7$	
	if	R_k is true
		$Array_k[A] = TD1$
		$LRU[A]$ is modified by rule M_k
		other blocks are not affected
		Stop the consequent check here.

To verify that the LRU algorithm is correctly implemented as specified, the assertion in Table 4 is supplied. Note that in the consequent, all possible conditions for replacement are enumerated. It is also note that a later condition depends on the false of all conditions stated before. With symbolic index A, all array rows are checked in a single assertion. A appears in both antecedent and initialization since otherwise (if the index in antecedent is different from the index in consequent) the LRU part will never be involved.

In this Section, we have seen the assertions for several classic array designs. An assertion is a rule which a particular array operation should obey. In essence, assertion can be seen as a program which can be interpreted by the STE process. The antecedent (and the initialization) consists of a sequence of "actions" to set the array to some expected states, and the consequent comprises a set of (mutually exclusive) conditions which the resulting array states should satisfy. Usually, to completely specify an array, four to five assertions are required. Note that with symbolic controls, more than one operations can be specified in a single assertion. For example, in the simple Tag example, setting "*sram_mode*" to a symbolic value "*s*" for a read can combine both "static read" and "tag load" into a single assertion.

At our design center, a high level Array Specification Language is designed to facilitate assertion writing (and assertion test generation as well). Such an assertion can then be translated into a more detailed code in FL, the underline functional language understood by the STE process [12]. As we have seen, assertion specification which hides many implementation details describes an array in a perspective very different from that by the RTL view. While RTL view exists for simulation and synthesis, assertion view is more suitable for the needs of test and verification.

4. Assertion Test Generation

The basic idea to generate tests from assertion is simple: *For each condition in the consequent, a set of test patterns are generated according to what have been specified in the antecedent.*

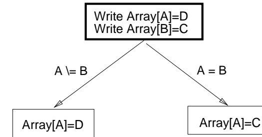

Fig. 8. Assertion tree for the simple array

We first consider the example in Figure 4 again. An *assertion tree* is constructed and shown in Figure 8. Then, we will generate tests to cover all paths in the tree. Note that A B C D are symbolic vectors. The simplest way is to fill up those symbols with randomly generated vectors. Hence, the tests can be the following.

Pattern 1:	(cover $A \neq B$)
	Write $Array[00] = 01$
	Write $Array[10] = 00$
	Read $Array[00]$
Pattern 2:	(cover $A = B$)
	Write $Array[11] = 10$
	Write $Array[11] = 00$
	Read $Array[11]$

Recall from Section 2 that these tests are used for three purpose: validating design, debugging errors, and detecting defects. As also mentioned there, since memory cells are tested by BIST already, for defect detection, tests should target only on those surrounding control logic that is not covered by the BIST. For this reason, the simple test sequence shown may or may not be enough for defects. For design validation, these tests are obviously not sufficient.

To extend the pattern set, we may enumerate all the possibilities in the address space. This requires $2^n \times 2^n$ patterns where n is the address bit length. This approach is very inefficient. To do a better job, extra test knowledge will be built in with the assertion tree so that instead of generating tests by a straightforward expansion of the tree from symbols to constants, a more sophisticated marching algorithm will be used. Suppose the address bit length is n (with 2^n memory locations). The modified marching algorithm looks like the following [2].

Table 5. Address Marching

For	$i = 0 \ldots 2^n - 1$
	$Array[i] \leftarrow$ unique random data d_i
For	$i = 0 \ldots 2^n - 1$ (1st)
	read $Array[i]$; $Array[i] \leftarrow d_i'$; read $Array[i]$
For	$i = 2^n - 1 \ldots 0$ (2nd)
	read $Array[i]$; $Array[i] \leftarrow d_i$; read $Array[i]$
For	$i = 2^n - 1 \ldots 0$ (3rd)
	read $Array[i]$; $Array[i] \leftarrow d_i'$; read $Array[i]$
For	$i = 0 \ldots 2^n - 1$
	read $Array[i]$; $Array[i] \leftarrow d_i$; read $Array[i]$

There are three levels of how the assertion tree can be expanded. The first level is a simple forward marching through all locations. This is enough for all decoder faults and cell stuck-at faults. The level two includes an additional backward marching to cover all cell state transition faults. Finally, the level three adds two more marchings starting with the complement values to cover coupling faults. Together with the simple random filling, there are four different ways to expand the assertion tree into tests. Depending on the purpose and the fault simulation results, we can choose one of them to most efficiently meet our needs. This simple assertion tree structure and its test generation approach serve as the basis of our test generation method described later.

The test patterns generated so far cover only the normal operation space. To address those tests beyond the normal space, we will construct another decision tree based upon the procedure which sets up control signals in the antecedent. For the simple array, to do a write, it involves setting WR_CLK to be "$0 \rightarrow 1 \rightarrow 0$" while keeping RD_CLK to be 0. The corresponding *control signal tree* is shown in Figure 9.

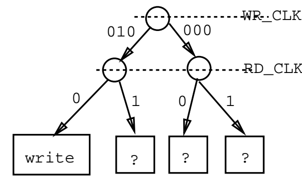

Fig. 9. Control signal tree from an assertion

Table 6. Tests for Abnormal Conditions

Pattern 1:	(set WR_CLK = 010, RD_CLK = 1)
	$Array[10] \leftarrow 01$
	$Array[10] \leftarrow 00$
	Read $Array[10]$
Pattern 2:	(set WR_CLK = 000, RD_CLK = 0)
	$Array[10] \leftarrow 01$
	$Array[10] \leftarrow 00$
	Read $Array[10]$
Pattern 3:	(set WR_CLK = 000, RD_CLK = 1)
	$Array[10] \leftarrow 01$
	$Array[10] \leftarrow 00$
	Read $Array[10]$

As it can be seen, we alter control signals for a normal operation in the following way. For a signal whose value remains as a constant during the operation, we complement the signal (such as RD_CLK). For a signal whose value changes, we replace each "change" with its "unchanged" value (such as WR_CLK). The second rule can be extended in many way. For example, if the normal signal is "0101010", then the "off side" signals can be "0001010" "0100010" "0101000" and "0000000." The key idea here is to delete a proper signal change and generate a test on it. In other words, we are testing the "necessary condition" for the operation. Hence, three more patterns will be generated for the three question marks in the Figure. Since the normal operation is a write, the random filling approach described earlier will be

used here for the three question marks. No additional marching is required. The three patterns are shown in Table 6.

The three patterns above provide a way to reach those test space beyond normal operations and hence, may detect wired defects/design errors which can not be seen during normal functional mode. However, some of the paths in the control signal tree may be illegal in the sense that they may cause unwanted behavior such as race, hazard, or oscillation. Therefore, all "side-path" vectors will be simulated with logic simulator (before fault simulation) to make sure that these vectors are valid.

In the following, we will extend the basic test generation techniques described so far to the three verification examples discussed in the previous section.

4.1. Static RAM

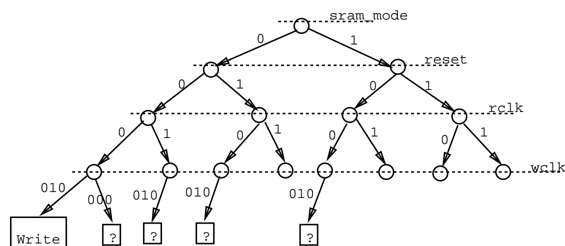

Fig. 10. Control signal tree for the assertion

The assertion tree in Figure 11 can be derived directly from Table 2 . By comparing Figure 11 to Figure 8, the static write tree here comprises four subtrees which match the structure in Figure 8. The four subtrees are selected by the condition symbol $S = s_0 s_1$. Therefore, one of the four different ways of generating the tests (one random and three marching) described above can be used to generate tests for each subtree. We arbitrarily choose level 2 marching for way 0 and 3, and level 1 marching for way 1 and 2. Also, in Figure 8, there is only one condition check in each leaf. Here, additional check to make sure that other blocks are not affected is necessary as specified in the assertion. This implies reading from other blocks every time after a new write is done. These additional reads can be injected for only one address location, all addresses in one pass of marching, or all addresses in all passes of marching. Again, this can be an option depending on the purpose and simulation results. The resulting tests can be the following (assume n-bit address space).

Table 7. Tests for Static Write

For	$i = 0 \ldots 2^n - 1$
	$Array_j[i] \leftarrow$ unique random data d_{j_i}
	for $j = 0, 1, 2, 3$
For	$i = 0 \ldots 2^n - 1$
	read $Array_0[i]$; $Array_0[i] \leftarrow d'_{0_i}$; read $Array_0[i]$
	read $Array_1[i]$; read $Array_2[i]$; read $Array_3[i]$
	read $Array_1[i]$; $Array_1[i] \leftarrow d'_{1_i}$; read $Array_1[i]$
	read $Array_0[i]$; read $Array_2[i]$; read $Array_3[i]$
	read $Array_2[i]$; $Array_2[i] \leftarrow d'_{2_i}$; read $Array_2[i]$
	read $Array_0[i]$; read $Array_1[i]$; read $Array_3[i]$
	read $Array_3[i]$; $Array_3[i] \leftarrow d'_{3_i}$; read $Array_3[i]$
	read $Array_0[i]$; read $Array_1[i]$; read $Array_2[i]$
For	$i = 2^n - 1 \ldots 0$
	read $Array_0[i]$; $Array_0[i] \leftarrow d_{0_i}$; read $Array_0[i]$
	read $Array_3[i]$; $Array_3[i] \leftarrow d_{3_i}$; read $Array_3[i]$

Note that if the above test sequence is too expensive. It can be reduced by not reading from the other three block for every i. Instead, a single i (say at 0) or a few i can be randomly selected. The key idea here is that the basic structure in Figure 8 will lead to the choice of a marching sequence, but any additional "side conditions" indexed by the symbolic address can be checked either at "a particular point of the march" or at "every address during the march", depending on the needs. It is not necessary that these side conditions be checked at every step.

The control signal tree is shown in Figure 10. The complete tree structure is expanded there. However, we may choose to generate tests for each question mark only. In other words, we allow to flip a control signal only one at a time and ignore multiple changes. Although it is possible to generate tests to cover the whole tree, Figure 10 demonstrates a more logical way to test only the necessary conditions. Also consider single stuck-at faults. It is more reasonable to flip a signal one at a time in order to detect the stuck-at faults than flipping multiple ones all together. The result-

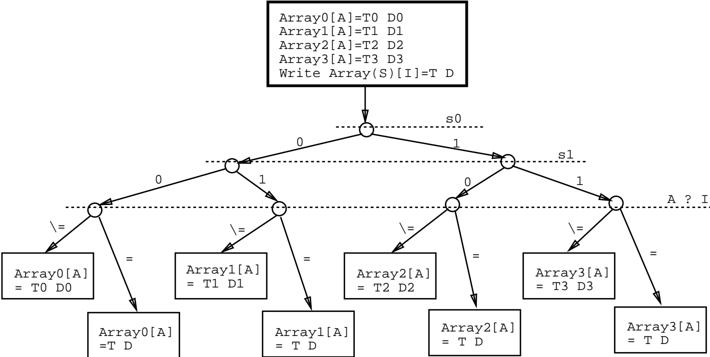

Fig. 11. Assertion tree for static write

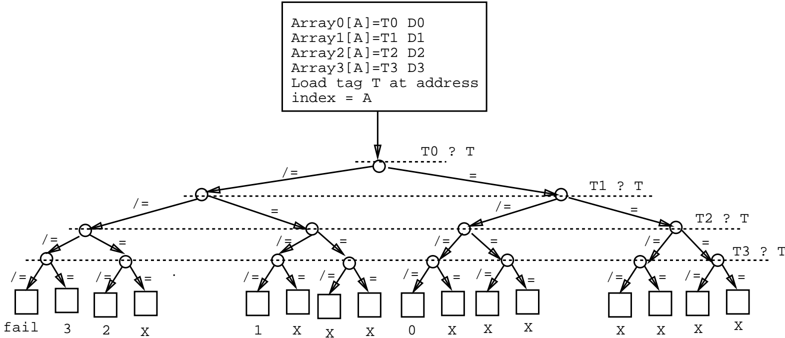

Fig. 12. Assertion tree for tag load

ing test pattern sequence will be similar to those shown earlier for the simple array example.

4.2. Tag Load

The assertion tree is shown in Figure 12. In this tree, there is only five leaves being defined. Others are considered to be impossible in the sense that surrounding logic will guarantee that those situations never happen for the tag. For those undefined leaves, we can choose either to ignore them or to generate a single test for each one of them by randomly filling up the symbols. This is straightforward.

For the remaining leaves, we explain a new idea called "data marching" in contrast to the "address marching" described earlier. In data marching, a symbolic comparison expression $T \neq T'$ will be filled up with the following sequence.

T	$000\ldots0$		$111\ldots1$
T'	$100\ldots0$		$011\ldots1$
	$010\ldots0$	and	$101\ldots1$
	$001\ldots0$		$110\ldots1$
	$\ldots$		$\ldots$
	$000\ldots1$		$111\ldots0$

On the other hand, $T = T'$ will imply only two tests: filling up T and T' with an arbitrary number and with its complement. The purpose of this data marching sequence is to detect all stuck-at faults in a comparator. Therefore, for the leaf labeled with "fail", we will fill T_0, T_1, T_2, T_3 all with $000\ldots0$ and $111\ldots1$, and supply T with the marching sequence. For the remaining four defined leaves, each will be given two patterns. For example, for label 3 leaf, $T_3 = 0000011\ldots1 = T$ in the first pattern and $T_3 = 1111100\ldots0 = T$ in the second. In both tests, other three tags can be filled with values other than that given to T_3.

Note that in the antecedent, only one symbolic address is used. Hence, there is no address march-

ing required. We can fill this symbolic address with a random number, a few selected addresses, or all possible values. The resulting tests can be the following (assume tag length m)

Table 8. Tests for Tag Load

Let $i =$	a single random address, or a few random addresses, or $0 \ldots 2^n - 1$
(for $k = 0, 1, 2, 3$)	
write tag $000 \ldots 0$, unique data d_{k_i} to $Array_k[i]$	
$j =$	$0 \ldots m-1$, tag load with tag $= 0 \ldots 010 \ldots 0$ where 1 appears in the jth bit;
write tag $111 \ldots 1$, data d'_{k_i} to $Array_k[i]$	
$j =$	$0 \ldots m-1$, tag load with tag $= 1 \ldots 101 \ldots 1$ where 0 appears in the jth bit
write unique tag t_{k_i}, data s_{k_i} to $Array_k[i]$	
tag load with tag $= t_{k_i}$, for $k = 0, 1, 2, 3$	
write tag t'_{k_i}, data s'_{k_i} to $Array_k[i]$	
tag load with tag $= t'_{k_i}$, for $k = 0, 1, 2, 3$	

The tests for control signal tree will be similar to those shown before and will be omitted.

4.3. LRU

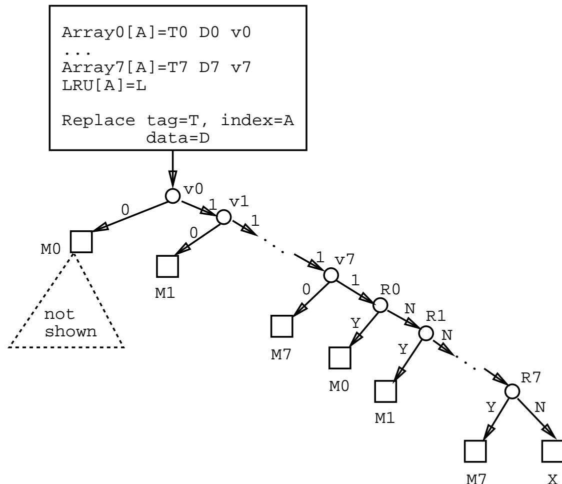

Fig. 13. Assertion tree for LRU

The assertion tree for LRU replace assertion in Table 4 is shown in Figure 13. The LRU example demonstrates the case where consequent contains complex conditions to capture complex control logic, but no symbolic address comparison or data comparison involved. Therefore, neither address marching nor data marching is required. Instead, each test pattern can be generated independently to cover a path in the assertion tree.

Table 9. Tests for LRU

Let $i =$	a single random address, or a few random addresses, or $0 \ldots 2^n - 1$
write to $Array_0[i] \ldots Array_7[i]$ with unique random Tags and Data $(TD)_0 \ldots (TD)_7$	
For	$j = 0 \ldots 7$ set valid bit 1 in $Array_0[i]$ to $Array_{j-1}[i]$ set valid bit 0 in $Array_j[i]$ to $Array_7[i]$ write 0000000 to $LRU[i]$ replace with $(TD)'_j$; read $Array_0[i] \ldots Array_7[i]$ and $LRU[i]$
write to $Array_0[i] \ldots Array_7[i]$ with unique random Tags and Data $(TD)_0 \ldots (TD)_7$ set all valid bits to 1	
For	$j = 0 \ldots 7$ $LRU[i] \leftarrow$ random LRU bits satisfying R_j replace with $(TD)'_j$ read $Array_0[i] \ldots Array_7[i]$ and $LRU[i]$
write to $Array_0[i] \ldots Array_7[i]$ with unique random Tags and Data $(TD)_0 \ldots (TD)_7$ set all valid bits to 1	
For	$j = 0 \ldots 7$ $LRU[i] \leftarrow$ random LRU bits satisfying NO R_j replace with $(TD)'_0$ read $Array_0[i] \ldots Array_7[i]$ and $LRU[i]$

First consider the $v_0 = 0$ branch. Although this branch leads to an immediate conclusion regardless of other conditions, we do have the choices of how to fill up the remaining conditions in the subtree following the branch (not shown). For example, we can randomly select a few values for $v_1 \ldots v_7$ and/or 'randomly fill up the LRU bits. Note that the LRU bits decide which rule R_i will be satisfied (at most one) and it is also possible that none of them does. The leaf labeled by "X" represents the case where LRU bits satisfies no R_i. This is usually considered illegal. However, we can choose to generate a pattern for this to test the unexpected branch. The resulting patterns are shown in Table 9 (Similarly, the address space A can be randomly enumerated).

Note that the assertion tree captures the functional intent of the control logic. Hence, although some randomness is involved for the data part while selecting the tests, the tests for the control logic remains rather deterministic. While mod-

eling issues may incur fault coverage loss for traditional ATPG approaches, the new assertion-tree based technique can still be more effective for random logic embedded in various array blocks.

4.4. Summary

The three examples above illustrates several basic ideas in the assertion test generation. We summarize the ideas below.

Address Marching For a boolean expression involving symbolic address comparison, one of the address marching algorithms will be used.

Data Marching For a boolean expression involving symbolic data comparison, a data marching algorithm will be used.

Control Logic For an assertion involving neither address nor data comparison, patterns are generated independently to cover all paths in the assertion tree.

Single Address Space For an assertion involving only one symbolic address, the address space are enumerated for a set of randomly selected ones.

Random Filling For other part of an assertion, usually random filling values are used.

These ideas are simple but sufficient to allow automatically generating tests from assertions for a wide range of array designs. Note that in all examples above, test generation involves using independent read and write procedures. Timing and control information for read and write can usually be extracted from their corresponding assertions (and the corresponding assertion procedures). From that, read and write "test procedures" for test generation can be pre-constructed before the actual test generation process starts.

Assertion specifications are translated directly into C programs which can be executed to produce test patterns based upon the techniques described. There are designs which do not perfectly fit into the test generation model. For example, content addressable arrays whose read operation is different from static RAM and whose assertion may require special encoding [11] requires more sophisticated analysis of the assertion tree. Another example is arrays which allow parallel read and write at a single clock cycle. This generates extra complexity for address marching. For these cases, manual tuning of generated patterns may be needed.

5. Results

For large industry designs such as microprocessor arrays, the single stuck-at fault model remains as the most efficient and practical approach to access test quality. While the excitation of manufacturing defects and design errors are unknown, complete stuck-at fault detections guarantee observation of all sites in a circuit, which is required to detect both defects and design errors. To assess the quality of assertion test, we first evaluate the tests with respect to stuck-at fault detection at the transistor level.

5.1. Transistor Level Stuck-at Fault Results

EXP1 The array is a simple static RAM with rather complex timing for speed optimization. An additional rule is imposed such that a write must precede with a read. Therefore, a write procedure now contains both read and write. This information can be extracted from the timing and control procedure associated with the assertion. Normal operations imply a read and a write assertions.

EXP2 The array is a tag with 4 way structures similar to that shown in Figure 5. Normal operations are static read/write and tag load.

EXP3 The array is a tag with 8 way structures plus an LRU control part as shown in Figure 7. Normal operations are static read/write, and tag load with LRU replacement. For this circuit, we focus on the LRU part.

Table 12 lists the number of faults in each experiment. Note that faults on the memory cells (tested by BIST) are not counted here. For all experiments, ATPG tests (which were generated from a commercial ATPG tool based on the gate level test views after reasonably amounts of DFT) are used for comparison. These tests achieve 95+% fault coverage for each design at the gate level test view. In all case, assertion tests are generated in an incremental way (adding more tests depending on the simulation results) but the final

Table 10. Normalized Results of Detected Stuck-at Faults

	Detected			
	Normalized Results		% of Detection By The Other	
	ATPG tests (D_1)	Assertion tests (D_2)	$(D_1 \cap D_2)/D_1$	$(D_1 \cap D_2)/D_2$
EXP1	1	1.24	99.7%	80.1%
EXP2	1	1.29	80.8%	67.2%
EXP3	1	1.07	99.6%	95.4%

Table 11. Normalized Results of Detected + Potentially Detected Stuck-at Faults

	Detected + Potentially Detected			
	Normalized Results		% of Detection By The Other	
	ATPG (DP_1)	Assertion tests (DP_2)	$(DP_1 \cap DP_2)/DP_1$	$(DP_1 \cap DP_2)/DP_2$
EXP1	1	1.23	99.9 %	81.1%
EXP2	1	1.85	100.0%	54.1%
EXP3	1	1.08	100.0%	93%

Table 12. Number of Faults

	Total Faults
EXP1	2309
EXP2	7930
EXP3	43290

test length does not exceed the size of the ATPG test set.

The faults are classified into three categories: Detected, Potentially Detected, and Undetected. Detected fault and undetected fault is defined as usual. A potentially detected fault is the one where in the faulty circuit, some outputs show the unknown value X while in the good machine they are constants (0 or 1). This usually occurs at the transistor level. Potentially detected faults can be caused by stuck-at faults at control lines. For example, a stuck-at zero at the write enable line to a cell will cause the cell content to be always X.

Results are presented in Table 10 and Table 11. In each table, the left two columns present normalized results with respect to the fault coverages from the ATPG tests. Take the result for EXP1 in Table 10. The 1.24 means that if the fault coverage from the ATPG tests is $f\%$, the assertion tests improve it to $1.24f\%$. The right two columns show the percentage of faults detected by one which is also detected by the other. For example, for EXP1 in Table 10, 99.7% of those faults detected by the ATPG tests are also detected by the assertion tests. Conversely, only 80.1% of the faults detected by the assertion tests are detected by the ATPG tests. Table 11 shows similar results based on both detected and potentially detected faults. Notice that in EXP2 and EXP3, all detected and potentially detected faults by the ATPG tests are detected or potentially detected by the assertion tests, but the reverse is not true. Therefore, assertion tests do provide extra fault detections in additions to those given by the ATPG tests. These results clearly demonstrate the superiority of the assertion tests.

5.2. Design Error Results

To evaluate the quality of assertion tests with respect to design error detection, experiments are performed to assess design error coverages based on the logic design error models in [1]. Preliminary results are also reported in [15].

For the experiments, an eight-way set associative tag array design is selected. The control logic surrounding the memory core consists of around 5500 gates. Since memory core is constructed in a regular way and is usually not a major concern for verification, design errors are injected only in the surrounding logic.

Two sets of 1023 design errors were randomly injected for the two experiments at the gate level,

Table 13. Results of Design Error Detection

	ATPG Tests	Assertion Tests
# of Test Vectors	1263	682
Design Errors Detected EXP I	728	965
Design Errors Detected EXP II	809	932

denoted as EXP I and EXP II. These design errors include extra inverter, extra wire, wrong wire, gate substitution, extra input, etc. as shown in [1]. When an injection is done, an error type is randomly picked. Design error coverages are obtained for ATPG tests and for the assertion tests.

Table 13 shows the comparison of design error detection by the two test sets. It is clear that the assertion tests outperform the ATPG tests. Note that all undetected errors are redundant. This can be proved by using the ATPG tool or by manually showing that it is impossible to detect the errors.

6. Conclusion

Test generation for embedded arrays has been a major challenge in test and verification of microprocessors. Due to the size and complexity of the array designs and the weakness of ATPG tools, extensive DFT works are required to avoid dramatic fault coverage loss. Sometimes, such attempts failed and DFT engineers had to decompose the design, generate partial tests, and manually combine them in order to achieve a very high coverage. The whole process was very tedious and inefficient. Moreover, the DFT process which involves modifying the gate level test view imposes an extra burden on the verification.

At Somerset, time to market is always a crucial factor to our success. Test generation for arrays was one of the bottleneck in our design process. To speed up the process, we had to develop a novel methodology.

In this paper, we have described a novel methodology to generate tests directly from the high level assertion specification. Assertion specification is originally for the purpose of STE formal verification, which are used to verify the correctness of both RTL view and schematic view.

By generating tests directly from the assertion specification, we eliminate the need to create gate level test view and the tedious DFT efforts. In addition, assertion tests are more functionally meaningful than ATPG tests with respect to design error debugging. To evaluate the quality of this new approach, we compare the results based on both stuck-at faults and logic design errors. Our experimental results demonstrate that assertion tests are superior to ATPG tests for both test (detecting defects) and verification (detecting design errors) purposes.

References

1. 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.
2. Magdy S. Abadir, and H. K. Reghbati, "Testing of random access memories," ACM Computing Surveys, September 1983.
3. Melvin A. Breuer and Sandeep K. Gupta, "Process Aggravated Noise (PAN): New Validation and Test Problems," International Test Conference, Washington FC., 1996., pp. 914-923.
4. 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.
5. Randal E. Bryant, "Symbolic simulation — techniques and applications," 27th Design Automation Conference, 1990.
6. Randal E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams," ACM Computing Surveys, Vol. 24, No. 3, Sep. 1992.
7. Neeta Ganguly, Magdy S. Abadir, and Manish Pandey, "PowerPC Array Verification Methodology Using Formal Verification Techniques," in Proc. International Test Conference, 1996. pp. 857-864.
8. C. Hunter, J. Slaton, J. Eno, R. Jessani, C. Dietz, "The PowerPC603(tm) Microprocessor: An Array Built-In Self Test Mechanism," in Proc. International Test Conference, 1994, pp. 388-394.
9. Charles H. Malley and M. Dieudonne, "Logic Verification Methodology for PowerPC Microprocessors," 32nd Design Automation Conference, 1995, pp. 234-240.

10. Manish Pandey, Richard Raimi, Derek L. Beatty, and Randal E. Bryant, "Formal verification of PowerPC^{TM} arrays using symbolic trajectory evaluation," in Proc. 33rd Design Automation Conference, 1996.
11. Manish Pandey, Richard Raimi, Randal E. Bryant, and Magdy S. Abadir "Formal verification of Content Addressable Memories Using Symbolic Trajectory Evaluation," in Proc. 34rd Design Automation Conference, 1997.
12. 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.
13. 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.
14. Li-C. Wang and Magdy S. Abadir, "A New Validation Methodology Combining Both Test and Formal Verification for PowerPC^{TM} Microprocessor Arrays," in Proc. International Test Conference, 1997, pp. 954-963.
15. Li-C. Wang, Magdy S. Abadir, and Jing Zeng, "On Logic and Transistor Level Design Error Detection of Various Validation Methods for PowerPC^{TM} Microprocessor Arrays," in Proc. VLSI Test Symposium, 1998, pp. 260-265.
16. PowerPC^{TM} 603e RISC Microprocessor User's Manual, Motorola Semiconductor Technical Data 1995.

Li-C. Wang received the B.S. degree with honors in Computer Engineering from National Chiao-Tung University, Taiwan in 1986, the M.S. degree in Computer Sciences in 1991 and Ph.D. degree in Computer and Electrical Engineering in 1996, both from the University of Texas at Austin. Currently, he is a member of the tool and methodology technical staff at Somerset PowerPC Design Center, Motorola, Inc. Prior to that, Dr. Wang worked at the Mathematics Research Center of Bell Labs, Murray Hill, New Jersey for the summers of 1991 to 1995. Dr. Wang's research interests lie in the areas of design for testability, high-level test generation, design validation, and formal verification.

Magdy S. Abadir received the B.S. degree with honors in Computer Science from the University of Alexandria, Egypt in 1978, the M.S. degree in Computer Science from the University of Saskatchewan, Saskatoon, Canada, in 1981, and the Ph.D. degree in Electrical Engineering from the University of Southern California, Los Angeles, in 1986. Currently he is Manager of the Test and Logic Verification Methodology and Tools group at Motorola's PowerPC Design Center (Somerset) in Austin, Texas. Prior to that he was the General Manager of Best IC Labs in Austin Texas (a Burn-in and Test Engineering firm). From 1986 to 1994 he worked at the Microelectronics and Computer Technology Corporation (MCC) as a senior member of the technical staff. Dr. Abadir has co-founded and chaired a series of international workshops on the economics of design, test and manufacturing. He has co-edited three books on that subject, and he also published over 60 technical journal and conference papers in the areas of test economics, design for test, computer-aided design, high-level test generation, and design verification and economics.