

Experience In Validation Of PowerPCTM* Microprocessor Embedded Arrays

LI-C. WANG AND MAGDY S. ABADIR

*Somerset PowerPC Design Center, Motorola Inc.,
6200 Bridgepoint Parkway, Bldg 4, Austin, Texas 78730
licwang@ee.tamu.edu
abadir@ibmoto.com*

; Revised May 10, 1999

Editor: Michael Nicolaidis and Bob Roy

Abstract. Design validation for embedded arrays remains as a challenging problem in today's microprocessor design environment. Although several methods for validating embedded arrays have been proposed [[5], [8], [9], [12], [14]], not much has been done to characterize the strengths and weaknesses of these methods. This paper provides a comprehensive study of various design validation approaches adopted at the Somerset PowerPC Design Center in the past, including methods from both formal verification and test generation. Effectiveness of these approaches will be measured based on automatic design error injection and simulation at both gate and transistor levels. Experience of using different validation approaches on recent PowerPC microprocessor arrays will be analyzed and discussed.

Keywords: Array, Logic Verification, Design Error, Symbolic Trajectory Evaluation, Assertion, Assertion Test Generation

1. Introduction

Embedded arrays (such as cache, cache tag, lookup tables, register files, etc.) in a microprocessor are complex sequential logic blocks which represent a major challenge in modern RISC 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.

In today's RISC microprocessors, embedded arrays may contain up to 80% of the transistors in use [5]. These arrays are usually custom designed so that performance can be optimized to the greatest extent. Hence, their validation becomes increasingly important. Traditionally, validating these arrays relied on logic simulation of vectors which are either manually generated or generated via automatic test pattern generation tools/methods. In the past few years, a number of new methods were proposed, including various methods from formal verification [[5], [8], [9], [14]] and a method from verification test generation [12]. While each method achieved certain success for validating arrays, not much has been done to characterize and quantify the strengths and weaknesses of these methods.

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

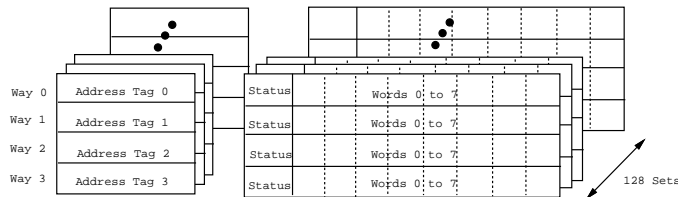


Fig. 1. PowerPC 603e Instruction Cache Organization

This paper presents a comprehensive study of various design validation methods for PowerPC microprocessor embedded arrays. These methods include:

- Simulation using tests generated by a commercial ATPG tools
- Formal verification based on Symbolic Trajectory Evaluation (STE) with manually created assertions [[5] [8], [9]]
- Formal verification based on STE with automatically generated assertions from RTL view [14], and
- Simulation using tests generated based on STE assertions [12].

Effectiveness of each method will be measured via systematic design error injection and simulation at both gate and transistor levels [13]. Experience of using these validation methods on recent PowerPC microprocessor arrays will be analyzed and discussed.

The paper is organized as follows. In Section 2, we provide a review of microprocessor arrays and design methodology. Also, we identify the problem of validation for arrays. Section 3 gives an overview of various design validation approaches mentioned above. Readers who are interested in specific aspects of those methods should refer to the previous papers for further detail. In Section 4, we discuss the strengths and weaknesses of those methods from our experience. Section 5 presents the systematic approaches for design error injection and simulation. Section 6 summarizes several typical experimental results and Section 7 concludes the paper.

2. Background

Embedded arrays in PowerPC microprocessors are varied and complex. They can be as simple as reg-

ister 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 603eTM microprocessor [15]. 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. Surrounding logic may use from 25% to 60% of the transistors in an array, depending on the complexity of its operation. 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 our design flow, an array design can be represented by three different views. They are:

- the high-level (or RTL) view
- the gate-level view, and
- the transistor-level view.

These 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.

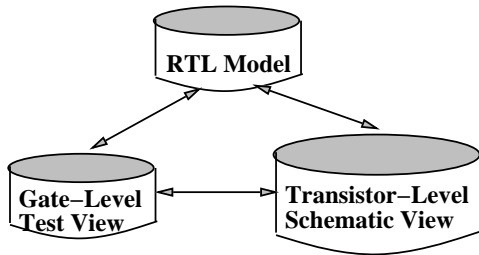


Fig. 2. Design Data Triangle

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. The constructs used in the RTL are all synthesizable and hence, they include typical if-then-else statement, case statement, logical operators (bit-wise or vector-wise), as well as arithmetic operators, etc. For-loop statement is allowed and is treated for macro expansion only. Usually, the RTL specification for an array does not involve any arithmetic statement.

Array designs are usually clock-based. Operations are defined based on a few (two system clocks plus scan clock, etc.) non-overlapping clocks. At the transistor level, dynamic logic is widely used. For custom design blocks such as arrays, transistor-level schematics are manually 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 the 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.

Logic validation is to verify the logical equivalence between the design specification (the RTL view) and the design implementation (the schematic view). Equivalence is based on the clocks and is defined for each individual operation (such as a read, a write, a tag load, etc.). In the design data triangle in Figure 2, all three views should be equivalent in that sense. In logic validation, however, properties such as delay, power consumption, and noise level are not verified. Verify-

ing those requires further simulation at the physical (layout) level.

3. Overview of Various Validation Approaches

Traditionally, vector simulation is used for logic validation. Due to the exponentially large number of patterns required for exhaustive validating the arrays, vector simulation is always questionable with respect to its completeness. Traditional boolean equivalence checking is not completely suitable for arrays either because of the complex timing scheme involved and the mismatch among latches in different views [7].

3.1. ATPG Based Test Vectors

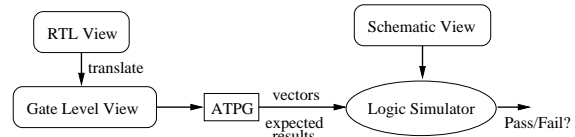


Fig. 3. Validation Based On ATPG Vectors

One method that we have employed in the past to verify the equivalence between the gate level and the transistor level views relies upon the use of ATPG test vectors generated on the gate level view and simulated on the transistor level schematic view [5] (see Figure 3.1). Usually, ATPG generated scan tests are used to cover all logic surrounding the memory cells because core array elements are tested by Built-In Self Test (BIST) circuitry implementing certain marching algorithms [6]. In order for ATPG tools to understand the array functionality, proper gate level view is created. In the gate level view, arrays are modeled with predefined latch and RAM primitives. Such a modeling approach is limited.

For high speed microprocessor design, arrays are designed with complex timing scheme to optimize their performance. As a result, DFT engineers spend much of their time refining the test view models for arrays using the available primitives and schemes that can be understood by ATPG tools. This process can be iterative and tedious.

3.2. STE Assertions

Formal verification with Symbolic Trajectory Evaluation (STE) [11] provides another alternative for validation. In STE, a set of so-called *assertions* should be created first for an array design. Assertions represent a set of high level properties for which the design should satisfy *under normal operations*. Therefore, in the STE methodology 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 (see Figure 3.2).

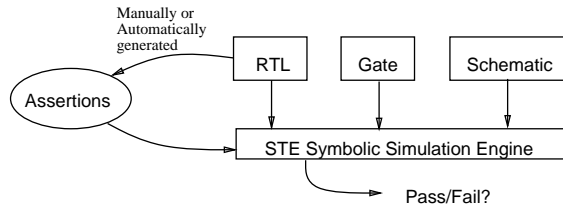


Fig. 4. STE Methodology

STE [11] is a descendant of symbolic simulation [3]. At the core of symbolic simulation, Ordered Binary Decision Diagram (OBDD) [4] 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 the assertions which are of the form “**Antecedent** \Rightarrow **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 [[5] [8] [9]].

With STE, operations specified in the antecedent are symbolically simulated, and then conditions declared in the consequent are asserted. At Somerset, an in-house tool called *VerSys* built

on top of *Voss* [10] is used to verify that all three views in Figure 2 satisfy the same set of assertions.

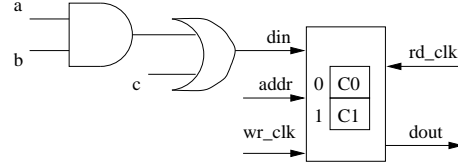


Fig. 5. A Simple Array Example

As a simple example, consider an array with only two 1-bit memory cells C0 and C1. There are 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 simplicity, consider only the write operation. 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. A “write” assertion can be very much like the following.

Initialization: $Array[Addr_0] = D_0$ at time 0
Antecedent: (Set controls to write)
 $a = A, b = B, c = C$
 and $addr = Addr_1$
 during time period $(0, T)$
 \Rightarrow (lead to)
Consequent: at time T ,
 if $(Addr_0 = Addr_1)$
 then $Array[Addr_0] = AB \mid C$
 else $Array[Addr_0] = D_0$

The power of such an assertion lies in the use of *symbolic indexing*, such as $Addr_0$ and $Addr_1$. With a symbolic address $Addr_0$, a simple reference in the consequent such as “ $Array[Addr_0]$ ” actually covers all cells in the array simultaneously.

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 [2]):

Table 1. Results of Symbolic Simulation

Let $D_1 := AB \mid C$
After Symbolic Simulation:
cell C0 = $(Addr_0 = Addr_1)[Addr'_0D_1 + Addr_0X] + (Addr_0 \neq Addr_1)[Addr'_0D_0 + Addr_0X]$
cell C1 = $(Addr_0 = Addr_1)[Addr_0D_1 + Addr'_0X] + (Addr_0 \neq Addr_1)[Addr_0D_0 + Addr'_0X]$

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.

For complex array designs, manual assertion creation can be a complicated, tedious, and unreliable process. First, a person needs to fully understand the functionality of a unit before creating the assertions. Assertions themselves can be long and tedious in order to specify all aspects of a design in detail. Often, a verification engineer would create assertions to verify only what he/she understood as the normal operations. Hence, assertions can be incomplete.

3.3. Automatic Assertion Generation for Arrays

Recently at Somerset, a novel way to automatically generate assertions from RTL view was deployed [14]. As we will see in the later section, this new approach, together with the power of STE, provides the most effective methodology for array verification.

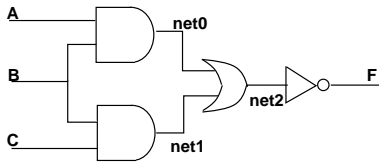


Fig. 6. A Simple Combinational Circuit

Automatic assertion generation for sequential designs in general is a difficult problem. Such a tool must be able to parse the statements in the RTL, automatically extract the high level properties of a design, and then express them in the antecedent/consequent pairs. On the other hand, generating assertion for a combinational design is rather straightforward. Consider the combinational logic shown in Figure 3.3. The assertion for this circuit can be stated as the following.

Antecedent:	\implies	Consequent:
$A = a, B = b,$		let $net0 = A \& B$
$C = c$		let $net1 = B \& C$
		let $net2 = net0 \mid net1$
		$F = \neg net2$

where $net0, net1, net2$ are temporary variables which hold the temporary symbolic expressions for the final consequent “ $F = \neg net2$ ”, and a, b, c are arbitrary symbols. Based on this simple observation above, the idea of our automatic assertion generation approach for arrays is to *abstract out the timing and sequential elements so that they can be treated as a combinational circuit at a given time*.

As mentioned in the Introduction, array designs are cycle-based, meaning that each operation is based upon a set of well-defined clocks. This allows us to divide the time of interest for verification into different phases where in each phase, the logic can be treated as a combinational circuit.

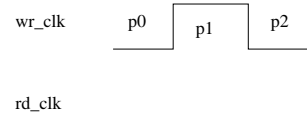


Fig. 7. Clocks For Write in the Simple Array

Consider the write operation for the simple array example above. Figure 6 depicts the corresponding clock scheme. Three phases are defined as $p0, p1$, and $p2$.

The RTL view for the simple array may look like the following.

```

Input a, b, c, addr, wr_clk, rd_clk;
Output dout;   Variable din;

din = ab | c ;
P-RAM Array(rd_clk, wr_clk,
            din, dout, Addr, 1);

End Module

```

In the RTL view, “P-RAM” is a predefined RAM primitive of which operations are well understood by designer and various tools. The “1” at the end of the P-RAM instantiation specifies that the array address is 1 bit only. The above example only illustrates the basic idea of using a RAM primitive to model the core memory. The syntax and usage are rather simplified and are not representative in general. In practice, a P-RAM primitive is much more complex, containing multiple read ports and write ports. Nevertheless, since it is impossible to include all functions in a primitive, anything not included should be modeled in the surrounding logic. For example, a memory may require that a write clock should be always given before a read clock. Such a constraint will be enforced through the surrounding logic instead of being modeled in the P-RAM primitive itself.

Using primitives for state-holding elements such as memory and latches greatly simplifies the process such as ATPG and automatic assertion generation since the behavior of a primitive is well defined and hence, a set of pre-determined processing rules can be associated with it.

Automatic assertion generation involves a process of translating each RTL statement into a set of temporary variables and boolean expression assignments during each clock phase. For example in the simple array case, let us assume that symbolic inputs should be specified only during clock phase “p1” which leave inputs during other intervals unspecified. Also assume that the memory is initialized with arbitrary data “ D_0 ” at arbitrary symbolic address “ $Addr_0$.” The translation results can be like the following.

$a_{[p0,1]} = X$, $a_{[p2,1]} = X$, $a_{[p1,1]} = A$,
 where A is an arbitrary symbol
 (similar results for b, c) (use symbols B, C)
 $din_{[p0,1]} = X$, $din_{[p2,1]} = X$,
 $din_{[p1,1]} = a_{[p1,1]}b_{[p1,1]} | c_{[p1,1]}$
 $addr_{[p0,1]} = X$, $addr_{[p2,1]} = X$, $addr_{[p1,1]} = Addr_1$,
 where $Addr_1$ is an arbitrary symbol

(Results:)

P-RAM $_{[p0,1,Addr_0]}$ = D_0
 P-RAM $_{[p1,1,Addr_0]}$ = $din(Addr_1 = Addr_0)$
 | $D_0(Addr_1 \neq Addr_0)$
 P-RAM $_{[p2,1,Addr_0]}$ = P-RAM $_{[p1,1,Addr_0]}$

In the above results, “ $A_{[p_i,cond]} = exp$ ” means that A at the end of phase p_i will hold a symbolic value given by the boolean expression exp when the $cond$ is true. Hence, $a_{[p1,1]} = A$ will be read as “input a at the end of clock phase p1 holds symbolic value A” where “1” in the condition part means “always.” $a_{[p0,1]} = X$ says that input a is unspecified. Similarly, “P-RAM $_{[p0,1,Addr_0]} = D_0$ ” will be read as “memory at the end of clock phase p0 hold data D_0 at location $Addr_0$.”

The last statement in the above results illustrates the idea of how to carry value across clock phases and thus, provides a way to simulate the sequential behavior of a state holding element. Note that the translation for a P-RAM element is based upon a set of pre-determined rules. Also note that inputs during phase 0 and phase 2 have no effect on the P-RAM content since the write clock is off. Detail can be found in [14].

For a write operation, consequences are formed at each state hold elements. Hence, the last three assignments for P-RAM will be treated as the assertion consequence. In general, consequences are also formed at all primary outputs as well.

The above example illustrates the basic ideas for automatic assertion generation from RTL. We summarize them below. Interesting reader can find further detail in [14].

- User first defines the operation clock and also specifies when to supply certain inputs.
- An assertion is constructed for each operation where operation clock is divided into a sequence of clock phases.
- For each clock phase, translation is done almost independently.
- Memory and latches are modeled using primitives. Rules of translation for these primitives are well defined.
- Each RTL statement is translated into a set of temporary variables and boolean expression assignments. Each variable is of the form $A_{[p_i,cond]} = exp$. Variables for a RAM primitive are of the form $RAM_{[p_i,cond,Addr]} = exp$.
- Consequences are formed at all state holding elements and all primary outputs for all clock phases given that their values are not X.

3.4. Assertion Based Test Vectors

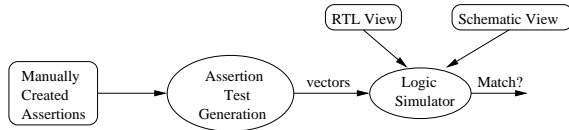


Fig. 8. Validation Based On Assertion Test Generation

Test generation based on STE assertion has been developed recently to combine the advantages of both ATPG method and STE approach. Figure 3.4 depicts its basic methodology. 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 [12].

The basic idea is to replace the symbolic values used in the antecedent with a set of constant test vectors (0 and 1) based on each condition specified in the consequent. Such replacements proceed by using proper test knowledge for testing common array logic, such as decoder, comparator, etc., in order to achieve high fault coverages with small numbers of tests. The basic techniques include (detail can be found in [12])

- replace a symbolic address comparison expression with *address marching* sequences (testing decoder)
- replace a symbolic data comparison expression with *data marching* sequences (testing comparator)
- replace stand-alone symbolic values with random vectors
- construct assertion decision tree and generate tests to cover all branches (testing control logic)
- construct control signal decision tree in order to generate tests to cover abnormal functional space (Usually, assertions specify what has been defined as the normal functionality)

4. Strengths and Weaknesses of Various Validation Methods

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 can be fortuitously detected [1], high fault coverage test set can be effective for detecting logic design errors. However, with respect to complete verification, even 100% fault coverage is still far from enough.

In STE methodology, because symbols are used in assertions, instead of 0 and 1, each assertion exhaustively verifies the functional aspect it targets on. With manually generated assertions, since they are produced independently from RTL view, not only the equivalence between two views with respect to the functionality of interest can be indirectly verified but also the correctness of RTL model can be checked again. This additional advantage is important if there is any unexpected error left in the RTL model after the extensive full-chip functional simulation.

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, completeness of verification is never guaranteed. Since the completeness of STE verification depends on the completeness of the corresponding assertions, it is possible that a particular design error will be missed due to the incompleteness of the assertion set.

Although STE can provide the most comprehensive verification for arrays, simulation based methods are usually 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 [2], delays and other detailed electrical properties (crosstalk, power, noise) are not verified during STE and hence, detailed simulation is required for further design validation.

For complex array designs, manually creating assertions can be a complicate, tedious, and unreliable process. Automatic assertion generation provides a systematic way to translate RTL view into assertions directly. Our experience indicates that this new approach is the most effective way for logic verification. Later, we will present experimental results to illustrate this.

The disadvantage of automatic assertion generation is due to its strict tight with RTL view. If there is an error in the RTL view, such an error will not be caught.

Finally, assertion test vectors 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 not as complete as the assertion. On the other hand, assertion tests do explore the abnormal functional space which is *generally not considered by assertions, and hence provide an 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 may be able to.

In a few large cases, when complete STE assertion verification becomes inefficiency due to the tremendous amount of memory required, assertion tests provide an indispensable alternative to assertions.

5. Error Models and Error Injection

None of the four 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 used for functional verification. For STE verification, since the underline symbolic simulator does not have the fault injection capability, fault simulation is not a feasible option.

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 strengths and weaknesses of various validation approaches in practice, and quantify how complete each approach can be.

5.1. Error Models in Gate Level View

Several design error models were developed at the logic level in [5]. These models include:

- Extra inverter
- Gate Substitution
- Extra wire
- Extra gate
- Missing gate
- Wrong wire, etc.

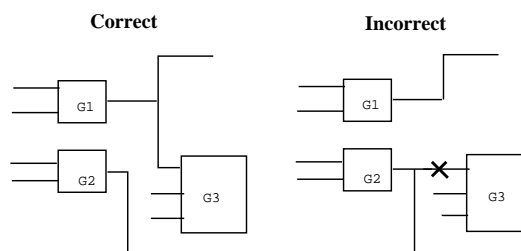


Fig. 9. Wrong Input Wire Design Error

To give an idea, Figure 5.1 shows an example of wrong wire design error [5]. These errors are injected to a design in the Gate Level Test View. Note that 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 injected 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 10 shows the modification for the extra inverter error model.

Thus with control input bus of width $\log(n)$, there are totally n possible input combinations where $n - 1$ can be used to select design errors. The modified design is then fed into a commercial logic simulator. Expected results from the original design are compared and if there is a mismatch, a detection is reported.

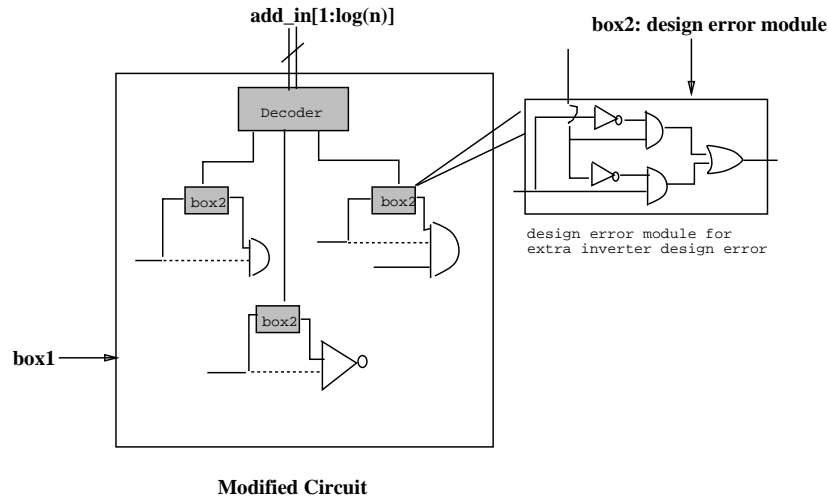


Fig. 10. Error Injection of an Extra Inverter

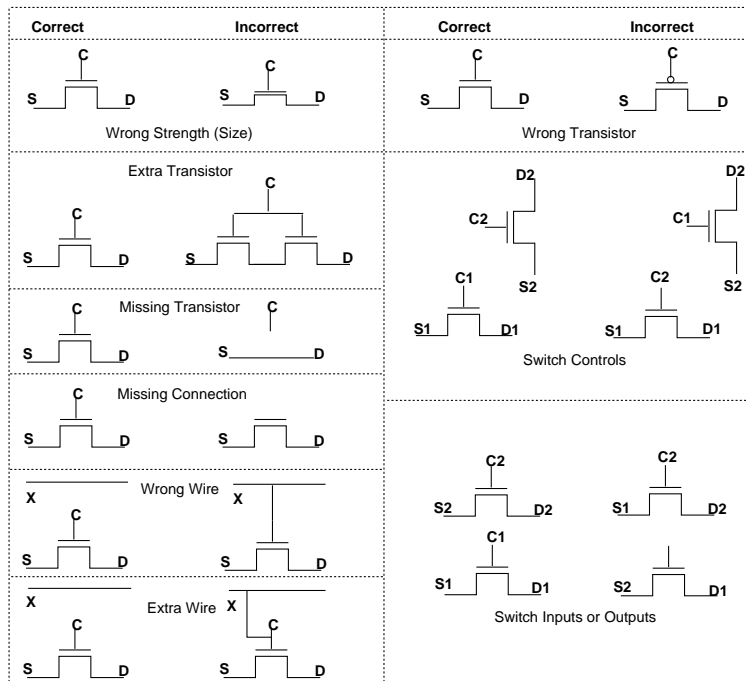


Fig. 11. Additional Error Models at the Transistor Level

5.2. Error Models in Transistor Level Schematic View

To accurately model the real design errors, different types of design errors are also injected in the transistor level schematic view. At the schematic level, a design may contain a hierarchy of mod-

ules within which both transistors and gates are used. Therefore, in addition to the gate level design errors described above, new types of hierarchical and transistor level design errors can be included.

Wrong Strength: Transistors can be categorized into different strength levels (as they

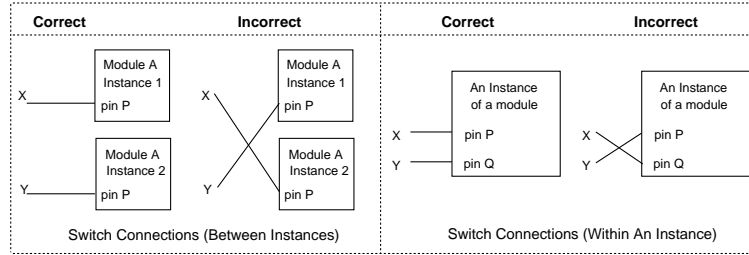


Fig. 12. Hierarchical Error Models

are turned on) based on their sizes (widths). Changing a transistor strength from one level to another may result in different circuit behavior. Our experience indicates that wrong strength is one of the most often seen errors that changes the logical behavior of the design at the transistor level. To catch this type of design errors, more detailed switch level simulation, such as the one used in STE methodology, is required.

Wrong Transistor: Switch an n-type transistor to p-type with the same size or vice versa.

Figure 11 illustrates the transistor level error models which are inspired by the gate level design error models. Although p-type transistors are shown in the Figure, similar error models apply for n-type transistors. Also note that missing transistor, missing connection, extra wire, and wrong wire error models can occur at transistor source and drain as well.

As mentioned earlier, schematic design contains hierarchy. A module in the lower level of the hierarchy is usually instantiated multiple times inside a module at the higher level. We thus consider additional two error types (see Figure 12).

- Switch connections within the same instance
- Switch connections between two instances of the same module

In addition, some gate level error models can also be generalized to module level and used in a hierarchical design. These include

- Wrong Wire
- Extra Wire
- Missing Wire
- Wrong Cell (For example, different implementations of the same function)

5.3. Error Injection

To inject errors at the transistor level, simply using a MUX design as before may not work properly. This is because a MUX designed at the gate level may change a weak signal at its input to be a strong signal at its output. To avoid such a situation, errors are injected and controlled by pass transistors which are given the largest width compared to all the transistors used in the original circuits so that when they are turned on, they do not change the transistor signal configuration in the original circuit. Figure 13 illustrates the idea. Note that with pass transistors multiple errors can be injected with one transistor easily. Also note that exactly one control should be 1 at a time, and when error is injected at other site, the “control to activate the good circuit” should be set to 1.

6. Results

For the experiments, an eight-way set associative tag array design is selected. This particular design is selected since it represents a typical practice in our daily validation effort. Although the results shown below may not hold for every array design, they are very much conclusive and consistent with our general experiences.

The control logic surrounding the memory core consists of around 5500 gates. As described earlier, memory cores are tested by BIST circuitry implementing certain marching algorithms [6]. ATPG tests are used to cover only the surrounding logic. Since memory cores are constructed in a regular way and are usually not a major concern for validation purpose, design errors are injected only in the surrounding logic as well. Area-wise, the surrounding logic occupies more than 50% of the design.

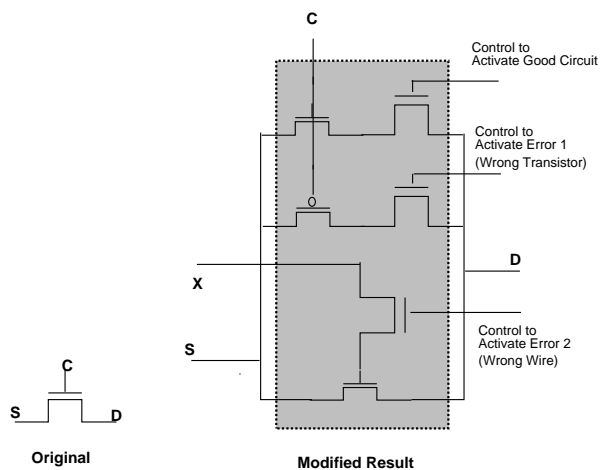


Fig. 13. Illustration of Error Injection at the Transistor Level

6.1. Gate Level Results

Two sets of 1023 design errors were randomly injected for the two experiments at the gate level, denoted as EXP I and EXP II. Error types are randomly selected and hence, results in almost equal number for each of the six types listed before. 10 additional primary inputs to the design are required. Design error coverages are obtained for all four approaches described earlier. Table 13 shows the stuck at fault coverage comparison between the ATPG tests and assertion tests, given that the fault coverage by assertion tests has been normalized to 1.

Table 2. Normalized Stuck-at Fault Coverages

	ATPG Tests	Assertion Tests
Fault Coverage	0.8	1
# of Test Vectors	1263	682

Table 3. Design Error Coverages

	ATPG Tests	Assertion Tests
EXP I	71.2%	94.3%
EXP II	79.1%	91.1%

Table 2 shows the comparison of design error detection by the two test sets. It is clear that the assertion tests outperform the ATPG tests. These results can be predicted from Table 13 since the design error models in use usually have a strong correlation to the stuck-at fault model [1]. Although it is not necessarily true, in general higher stuck-at fault coverage does imply a higher design error coverage.

Table 4 demonstrates the design error detection results by manually generated STE assertions. These assertions were generated by verification engineers with the helps from designers for understanding the array functionality. Note that in the Table an error may be detected multiple times by different assertions. It is also interesting to observe that these symbolic assertions do not outperform the assertion tests in number 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 as well*. Hence, a design error that changes the behavior of a circuit in the abnormal functional space will not be considered as a true design error by designer. Note that assertions for design errors in the abnormal functional space can also be achieved by adding more assertions to target on that space specifically.

Table 4 shows the results from automatically generated assertions. As we can see, by systematically applying the translation rules described ear-

Table 5. Design Error Coverages by Manually Generated Symbolic Assertions

	Assertion 1	Assertion 2	Assertion 3	Total
Errors Detected (EXP I)	27.6%	74.1%	36.5%	94/2%
Errors Detected (EXP II)	25%	69.3%	33.2%	90.8%

lier to obtain assertions directly from RTL view, error detection rates improve significantly.

Table 4. Design Error Coverages by Automatically Generated Symbolic Assertions

Error Coverages	
EXP I	96.8%
EXP II	94.3%

6.2. Transistor Level Results

Two sets of 511 errors (of different types) are randomly injected, one for the transistor level design errors and the other for the hierarchical design errors described earlier. This requires adding 9 primary inputs as error injection controls. The errors were injected at the module level in the hierarchical schematic design of the tag array. Table 6.2 shows the comparison results between assertion tests and ATPG tests. The results demonstrate that assertion tests outperform ATPG tests consistently. Note that for transistor level design errors, the detection rate is substantially lower than the gate level cases. At the transistor level, many of the injected design errors are redundant.

Table 6. Design Error Coverages at the Transistor/Hierarchical Level

	ATPG Tests	Assertion Tests
Transistor Level	56%	76.9%
Hierarchical Level	83%	96.1%

Table 6 presents the results from both manually generated and automatically generated assertions. The results are consistent with the gate level re-

sults except that detection rates for the transistor level design errors injected are substantially lower.

Table 7. Design Error Coverages by Manually/Automatically Generated Assertions

	Manual	Automatic
Transistor Level Errors Detected	76.5%	81%
Hierarchical Errors Detected	96.1%	96.1%

6.3. Summary

Figure 14 summarizes the comparison results in terms of the Venn diagrams.

- All design errors detected by the ATPG tests at the gate level were also detected by the assertion tests. However, this is not true for the stuck-at fault results shown in Table 13 and is not true for the case of transistor level/hierarchical design error either.
- For all cases, assertion tests and manually created assertions together can detect all errors captured by ATPG tests. Unlike the ATPG process, generating the assertion tests is not time consuming [12]. Hence, assertion tests can be seen as an inexpensive and relatively effective validation method. Also note that for large arrays, STE usually requires a large amount of memory to store the OBDDs. Therefore, a more feasible validation approach for those arrays can be applying assertion tests first and then, producing partial assertion (by reducing the number of symbols in use to reduce the OBDD size) to cover the remaining undetected errors.
- At the gate level, 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. At the transistor level, many errors can be shown redundant

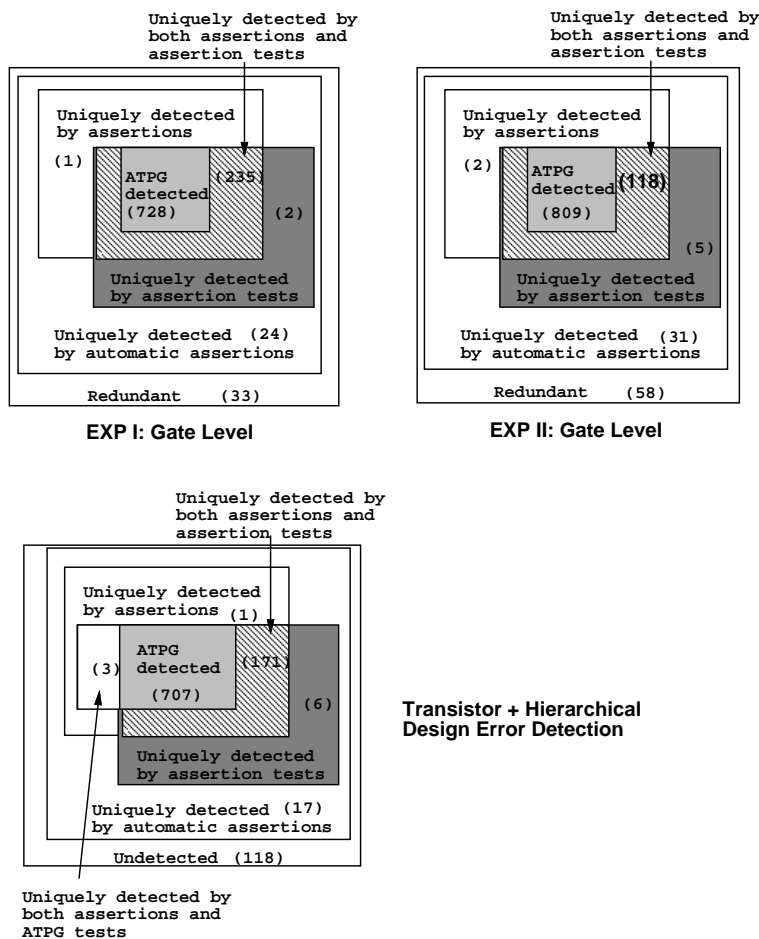


Fig. 14. Relations Among Errors Detected By The Four Methods

manually. Since the ATPG tool does not handle transistor level view, not all undetected errors can be proved to be redundant.

- Most significantly, automatically generated assertions can cover design errors detected by all other three methods, plus additional detections on its own. These results clearly demonstrate the effectiveness of this new approach for design validation.

However, two disadvantages should be noted. First, in the above experiments, the RTL view is treated as the golden model. Hence, if the RTL view is incorrect, only manually created assertions have the chance to detect that. Second, since automatically generated assertions do not intend to optimize OBDD memory usage as manually created assertions may do, in some cases (for example, for content address-

able memory, a special CAM encoding can be used [9]), the amount of memory required can be much larger.

- In Figure 14, for the case of transistor level design error, hierarchical error detection does not contribute much to the differences. This can be observed from the results shown before. Except for the ATPG vector simulation, all other methods detect the same set of hierarchical errors injected. Hence, the differences shown among different methods come mainly from transistor level error detection only.

7. Conclusion

In this paper, we study four different validation approaches, specifically for their effectiveness with

respect to design error detection. The results demonstrate that with respect to verification completeness, automatically generated assertions, together with the STE methodology, provides us the most effective approach. Nevertheless, manually created assertions do have the advantage of performing independent check on RTL view, and assertion tests are more applicable to large design when OBDD memory usage becomes a problem. Future research is required to produce assertion tests from automatically generated assertions, which may potentially provides a better tradeoff between the quality and the efficiency.

Our results also indicate that design error injection can be 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. By analyzing the undetected errors, the weaknesses of each method at any given stage of the design cycle can be identified and removed for the next stage.

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. 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.
3. Randal E. Bryant, *Symbolic Simulation — Techniques and Applications*, in Proc. 27th Design Automation Conference, 1990.
4. Randal E. Bryant, *Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams*, ACM Computing Surveys, Vol. 24, No. 3, Sep. 1992.
5. Neeta Ganguly, Magdy S. Abadir, and Manish Pandey, *PowerPC Array Verification Methodology Using Formal Verification Techniques*, in Proc. International Test Conference, Washington DC., 1996. pp. 857-864.
6. 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.
7. Charles H. Malley and M. Dieudonne, *Logic Verification Methodology for PowerPC Microprocessors*, in Proc. 32nd Design Automation Conference, 1995, pp. 234-240.
8. Manish Pandey, Richard Raimi, Derek L. Beatty, and Randal E. Bryant, *Formal Verification of PowerPCTM Arrays Using Symbolic Trajectory Evaluation*, in Proc. 33rd Design Automation Conference, Las Vegas, NV., 1996.
9. 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.
10. 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.
11. 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.
12. Li-C. Wang, and Magdy S. Abadir, *Test Generation Based on High-Level Assertion Specification for PowerPCTM Microprocessor Embedded Arrays*, Journal of Electronic Testing, No. 13, 1998, pp. 121-135.
13. Li-C. Wang, Magdy S. Abadir, and Jing Zeng, *On Logic and Transistor Level Design Error Detection of Various Validation Approaches For PowerPCTM Microprocessor Arrays*, in Proc. VLSI Test Symposium, 1998.
14. Li-C. Wang, and Magdy S. Abadir, *Automatic Generation of Assertions for Formal Verification of PowerPCTM Microprocessor Arrays*, in Proc. Design Automation Conference, 1998.
15. PowerPCTM 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 an assistant professor in the Computer Engineering Group of Department of Electrical Engineering at Texas A & M University, College Station. He also works part-time as a member of the tool and methodology technical staff at the Somerset PowerPC Design Center, Motorola, Inc. Prior to that, Dr. Wang worked at Somerset, Motorola for 3 years and 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.