Combining ATPG and Symbolic Simulation for Efficient Validation of Embedded Array Systems

Ganapathy Parthasarathy[†], Madhu K. Iyer[†], Tao Feng[†]
Li-C. Wang[†], Kwang-Ting (Tim) Cheng[†], and Magdy S. Abadir[††]

Department of ECE, UC-Santa Barbara[†]
ASP Advanced Tools and Methodologies, Motorola, Austin, Texas[††]

Abstract

In the past, Symbolic Trajectory Evaluation (STE) has been shown to be effective for verifying individual array blocks. However, when applying STE to verify multiple array blocks together as a single system, the run-time OBDD sizes would often blow up. In this paper, we propose the use of both ATPG-based justification engine and symbolic simulation to facilitate the application of STE proof methodology for array systems. Our method translates a given verification problem instance into ATPG justification objectives, and partitions a given design into ATPG and symbolic simulation domains. Then, by developing a scheme that enables ATPG justification engine to work closely with the symbolic simulator, the run-time OBDD sizes during each symbolic simulation run can be limited. We demonstrate the effectiveness of our approach by verifying the Memory Management Unit (MMU) in Motorola high-performance microprocessors. The verification of MMU as a whole was not possible before because of the OBDD size blow-up problem when symbolic simulation is used in the STE proof process.

1 Introduction

Embedded memories or *arrays* are important components in digital ICs for most high-performance applications. The speed of these on-chip memory components is critical to the overall performance of the chip. They are often custom-designed at the transistor level to optimize the performance. Dynamic logic and self-time circuitry are widely used in these designs. Pipelining and prediction techniques are commonly employed. A typical array system contains multiple array blocks. Interactions among these blocks can be complex and hard to verify.

The size, complexity, and sequential nature of embedded memories make their test and validation very challenging. Because they are custom designs, correct modeling can be time-consuming and error-prone. Both simulation-based and formal techniques have been tried to validate arrays. However, simulation-based approaches are often incomplete due to the exponentially large state space. Formal techniques such as STE assertion-based symbolic simulation have been shown to be successful in terms of both functional verification and structural equivalence. Nevertheless, these techniques could only be applied for verifying individual array blocks [1, 2]. In STE proof method, symbolic simulation is used as the underlying engine [3, 4, 5, 7, 6, 8, 9, 10] which utilizes *Ordered Boolean Decision Diagrams* (OBDDs) [11] to represent ternary logic functions.

The OBDD-based STE proof method works well for each individual array [12, 13]. With special symbolic encoding techniques, it can also be efficient to verify arrays which are content addressable in nature [14]. Nevertheless, when verifying an operation that involves interactions among multiple array blocks, the OBDD sizes often blow up. Therefore, in the past the STE method was never applied at the system level where multiple blocks are combined as a whole.

In this work, we attempt to provide a solution to avoid the OBDD size blow-up problem so that STE proof method can be applied at the system level. Our approach is to partition the problem into ATPG and symbolic simulation domains, and to utilize both ATPG justification engine and OBDD-based symbolic simulator to solve the problem. By developing an efficient scheme that enables both engines to work together, the OBDD sizes during each symbolic simulation run can be limited. Our goal is to demonstrate that this new strategy can verify multiple array blocks together as a single system, as opposed to verify individual blocks separately in a system.

The rest of this paper is organized as follows: Section 2 reviews prior related work and points out the novelty of our approach. Section 3 describes the STE, the assertion proof methodology, and OBDD-based ternary symbolic simulation. In Section 4, we use a content addressable memory as an example to illustrate the problem of symbolic simulation, and illustrates the fundamental idea to combine ATPG and symbolic simulation. The detail of our techniques are presented in Section 5. Section 6 includes a sequence of experiments to compare symbolic simulation to the proposed ATPG/symbolic simulation combined approach. The effectiveness of our method will be demonstrated based upon these experiments using Motorola microprocessor MMU design. Section 7 concludes the paper.

2 Comparison to Prior Work

STE has been used to validate memory arrays such as on-chip caches and register files. In [12], authors used the *Voss* STE [15] system to verify a multi-ported register file unit and a data tag unit in *PowerPC* ^{TM} microprocessors. With *symbolic indexing*, the number of variables required to verify properties of these arrays was approximately logarithmic in the number of memory locations, thus ameliorating the state explosion problem. In [14], authors introduced the usage of special symbolic encoding schemes to minimize the OBDD size complexity of STE verification for content addressable memories (CAMs). Two CAMs, a Block Address Translation (BAT) unit and a Branch Target Address Cache (BTAC) unit, were verified with the new schemes. For all these circuits, the verification can be done at the switch level, so that the validation was performed on the actual designs. In [2], various validation methods, including STE were compared based upon design error injection and simulation. It was shown that although STE approach was able to validate array blocks, the quality of validation could be highly dependent on the *assertions* supplied to the STE process. Therefore, to ensure high quality, automatic generation of assertions [16] should be used. In a recent work [13], STE was employed as an automatic structural equivalence checking methodology (between RTL and schematics) for Motorola high-performance microprocessor arrays.

All of the previous work utilizes STE to target individual blocks of an array system. Due to OBDD size complexity, STE was never applied at the system level to verify multiple array blocks as a whole. In the past, when verifying an individual array block, block input constraints imposed by the outputs of other blocks were extracted manually. In many verification cases, including both equivalence checking and (block-level) functional verification, these input constraints can greatly affect the resulting quality. To ensure complete verification, it is imperative to verify multiple array blocks together as an unified system.

For test applications, verifying block input constraints is crucial as well. Valid input constraints can help to remove scan patterns that are not achievable in full-system functional mode. In many cases, especially for custom arrays, application of non-functional patterns in scan may result in unexpected problems on the actual silicon (such as power problem).

In this work, we propose to combine ATPG justification engine and OBDD-based symbolic simulator for array system validation. The idea of using multiple engines (such as combining SAT solver and OBDD-based symbolic simulation) is not new. For example, Wilson *et. al.*, [17, 18, 19] proposed using a combined SAT and OBDD-based symbolic simulation approach to eliminate run-time OBDD memory blow-up. Huang *et. al.*, [20] proposed using arithmetic and Boolean ATPG solver to verify arithmetic circuits. Hazelhurst and Seger have explored a hybrid approach based on theorem-proving and STE [21]. STE is used to prove low-level properties of the circuit. A set of inference rules are used to compose the results of STE in a theorem proving environment. The hybrid approach was able to verify a 64-bit multiplier by verifying the individual adders using STE and then composing the results to show that the adders were properly connected. Aagaard and Seger used the hybrid approach to verify a radix-eight, pipelined, IEEE double-precision floating point multiplier [22].

The key difference of our work from prior work is that our approach is designed to optimize the run-time efficiency for verification problems that involves multiple array blocks. We will describe a novel scheme that enables ATPG and symbolic simulation to efficiently work together. The effectiveness of our techniques will be demonstrated through experiments on an MMU design from Motorola high-performance microprocessors [23].

3 Background

Symbolic Trajectory Evaluation (STE) [8, 9] is a formal verification technique that is based on ternary symbolic simulation [6, 7]. *Ordered Binary Decision Diagrams* (OBDDs) [11] are used to efficiently manipulate ternary functions during STE.

In STE, specifications are given as *trajectory assertions* of the form **Antecedent** $\Longrightarrow_{LEADTO}$ **Consequent** where both *Antecedent* and Consequent consist of *trajectory formulae*. Assertions specify a set of design properties in a restricted temporal logic form. STE checks to see if a given design satisfies a set of given assertions.

A trajectory formula can be a simple predicate such as "$node_i$ is 0" which specifies that the signal $node_i$ should contain the value 0 at the present time. With conjunction, case restriction, and the *next time* operator, trajectory formulae can be constructed from the simple predicates. Moreover, there is a domain restriction operation *when* allowing specification of input constraints in the antecedent and restricted output results in the consequent. If V is a trajectory formula, and D is a Boolean function, "$node_i$ is V *when* D" specifies that only when D is true, $node_i$ contains the value V. Such a specification usually has two semantic meanings depending on when it is placed. If it appears in the antecedent, then for unspecified domain D', the value of $node_i$ is the don't care "X." If it appears in the consequent, then the computed value of $node_i$ from the symbolic simulation of the circuit will only be checked under the domain D. The value of $node_i$ under the domain D' is ignored.

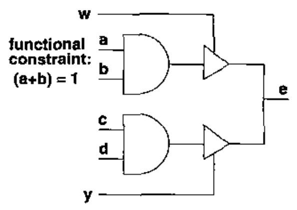


Figure 1: An Illustrative Example

Figure 1 gives a simple circuit example. Notice that the inputs a and b are restricted by a functional constraint. Such a constraint can be from the surrounding logic not shown in the figure. The following describes a valid assertion to be checked (where A, B, C, D, W, Y are arbitrary 1-bit symbols). Note that we use "$\wedge$" and "$\vee$" to specify the conjunction and disjunction of formulae in the assertion, and we use "$\cdot$" and "$+$" to specify logic AND and logic OR in Boolean function.

$$(when(A = 1 \vee B = 1)(a = A)) \wedge$$
$$(when(A = 1 \vee B = 1)(b = B)) \wedge$$
$$(c = C) \wedge (d = D) \wedge (w = W) \wedge (y = Y)$$
$$\Longrightarrow_{LEADTO}$$

$$(when((W = 1 \wedge Y \neq 1) \wedge (A = 1 \vee B = 1))(e = A \cdot B)) \wedge$$
$$(when(Y = 1 \wedge W \neq 1)(e = C \cdot D))$$

The antecedent in the above assertion supplies symbolic inputs. In the STE process, ternary symbolic simulator simulates the given inputs and input constraints on the circuit. The results are compared with the consequent. Note that the consequent part checks the results only when $W \neq Y$. When $W = Y$, the output e is defined as unknown value "X" and hence, is not checked. Also note that the domain constraints can be applied in both antecedent and consequent.

Depending on how delay models are defined, the notation "$\Longrightarrow_{LEADTO}$" in the assertion can have multiple semantics. If zero delay model is used, "$\Longrightarrow_{LEADTO}$" behaves like a simple implication. If 1 unit delay is associated with every gate, "$\Longrightarrow_{LEADTO}$" suggests that the consequent is true after 2 time units.

When symbolic simulation is carried out directly on the transistor-level models, unit delay model is necessary for capturing the behavior of dynamic logic and self-time circuitry [13]. If simulation is done on RTL or gate-level models, zero delay can be used. In the later case, the symbolic simulator behaves like a cycle-based simulator.

3.1 STE as Cycle-Based Constraint Solving

In our work, we assume that the symbolic simulator is cycle-based, and the assertion for an array unit (which may contain multiple array blocks) is given by the user. The goal is to check that the RTL model satisfies the given assertion.

Conceptually, trajectory formulae specified in both antecedent and consequent are *constraints* on circuit signals. These constraints are ternary logic functions. For two ternary functions $f_1(x_1 \dots x_n), f_2(x_1 \dots x_n)$, we say that f_2 *satisfies* f_1 if the following two conditions hold:

1. $\forall$ binary values $v_1 \dots v_n$ assigned to $x_1 \dots x_n$, if $f_1(v_1 \dots v_n) = 1$, then $f_2(v_1 \dots v_n) = 1$.
2. $\forall$ binary values $v_1 \dots v_n$ assigned to $x_1 \dots x_n$, if $f_1(v_1 \dots v_n) = 0$, then $f_2(v_1 \dots v_n) = 0$.

Note that because f_1 is a ternary function, it is possible to have an input assignment $v_1 \dots v_n$ such that $f_1(v_1 \dots v_n)$ =X. For those inputs, $f_2(v_1 \dots v_n)$ is unrestricted, and can be one of the "0, 1, and X."

In the context of above ternary satisfiability check, the problem of checking the assertion in STE can be viewed as a *constraint solving* problem. This is illustrated in Figure 2.

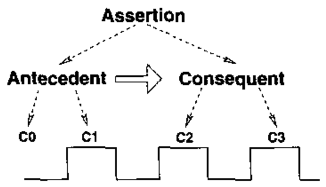


Figure 2: Constraint Solving in STE

In a cycle-based simulation environment, an assertion imposes ternary logic constraints on circuit signals based upon clock definition. For the circuit to satisfy the assertion, it suffices to check to see if all constraints can be satisfied simultaneously. When a constraint given by the antecedent cannot be satisfied, this represents the "over-constrained" situation (meaning that the input constraints are inconsistent) defined in the STE [15]. If a constraint given by the consequent cannot be satisfied, then it indeed represents an assertion fail.

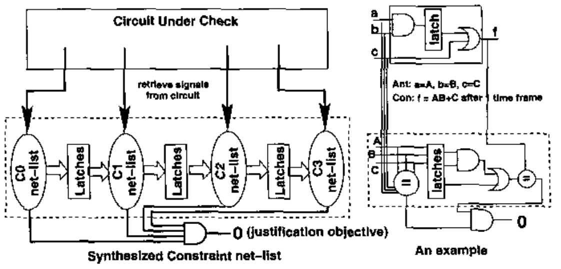


Figure 3: Constraint Solving as ATPG Justification

The problem of simultaneously satisfying all constraints can be modeled as an ATPG justification problem as illustrated in Figure 3. In such a formulation, constraints are synthesized into constraint circuitry, and the task of ATPG is to find a test vector to make the output of the justification AND gate equal to 0. We note that when the input space is limited by the domain constraints in the antecedent, the ATPG search space is restricted. Therefore, the test vector can only come from the functional domains specified by the antecedent constraints.

Since symbolic simulation utilizes OBDDs, and ATPG utilizes backtracking search, by converting a STE assertion check problem into an ATPG justification problem, we essentially transform the space complexity into time complexity. Therefore, if an ATPG justification engine is used to solve the problem, and if in the original problem the OBDD sizes would blow up, then in the transformed problem, the ATPG run time will most likely blow up as well. Hence, applying ATPG alone to the constraint solving problem will not be sufficient.

4 CAM: An Illustration Example

In this section, we use a simple content addressable memory (CAM) to illustrate why symbolic simulation or ATPG alone can be inefficient. Figure 4 depicts a CAM example similar to the illustration example used in [14].

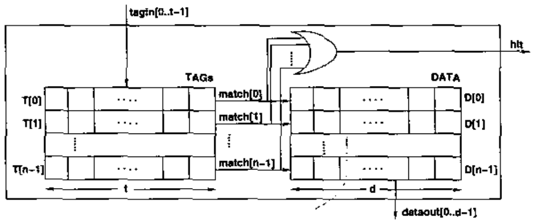


Figure 4: A CAM Example: Tag size = t, No. of entries = n, and Data size = d

The CAM performs the following operation. The inputs $tagin[0..t-1]$ are compared with n tag entries

$T[0], T[1], \ldots, T[n-1]$. If $T[i]$ matches the *tagin*, then $match[i] = 1$ and the corresponding data $D[i]$ are placed at the output $dataout[0..d-1]$. For all other mismatch entries j, $match[j] = 0$. Normally, we assume that at most one tag entry can match the incoming tag. The *hit* signal indicates that there is a match in the tag.

Assume that $n = 4$. Let $T^{vec}, T_0^{vec}, T_1^{vec}, T_2^{vec}, T_3^{vec}$ be t-bit symbolic vectors. Let $D_0^{vec}, D_1^{vec}, D_2^{vec}, D_3^{vec}$ be d-bit symbolic vectors. Then, the CAM operation can be specified as the following assertion.

$$\begin{gathered}(T[0] = T_0^{vec}) \wedge (T[1] = T_1^{vec}) \wedge (T[2] = T_2^{vec}) \wedge (T[3] = T_3^{vec}) \wedge \\ (tagin = T^{vec}) \wedge (D[0] = D_0^{vec}) \wedge \\ (D[1] = D_1^{vec}) \wedge (D[2] = D_2^{vec}) \wedge (D[3] = D_3^{vec}) \\ \Longrightarrow_{LEADTO} \\ (when(nomatch)(hit = 0)) \wedge \\ (when(matchonly0)(hit = 1 \wedge dataout = D_0^{vec})) \wedge \\ (when(matchonly1)(hit = 1 \wedge dataout = D_1^{vec})) \wedge \\ (when(matchonly2)(hit = 1 \wedge dataout = D_2^{vec})) \wedge \\ (when(matchonly3)(hit = 1 \wedge dataout = D_3^{vec}))\end{gathered}$$

where *nomatch* indicates the condition that $T^{vec} \neq T_i^{vec}$ for all $i = 0, 1, 2, 3$. And *matchonly0* indicates the condition that T^{vec} matches only T_0^{vec} (and so on).

When OBDD-based symbolic simulation is used to verify the assertion, the OBDD size can easily blow up at the output *hit* [14]. This is because the OR function for generating the *hit* signal creates too much interdependency among the symbolic tag variables used in the assertion, and the number of tag variables is large ($= t \times n + t$).

If the problem is transformed into an ATPG justification instance and an ATPG is used, the run time can exponentially blow up as well. This can be illustrated by considering the equivalence checking problem of two t-bit comparators. The ATPG would make $O(2^t)$ decisions to exhaust the search space. For CAM design, it is not unusual to have $t > 32$. Hence, ATPG alone cannot solve the problem.

4.1 Split Assertion by ATPG

One possible way to solve the problem is to partition the design into ATPG and symbolic simulation domains. An intuitive partition is to give the OR gate for generating the *hit* signal to ATPG, and leave the rest of the design to symbolic simulation. Then, suppose that ATPG intends to justify a "0" on the *hit* signal. By ATPG implication, it will assign "0" to all internal $match[i]$ signals (for $i = 0 \ldots n-1$). Then, ATPG can call the symbolic simulation to check if indeed $match[i] = 0$ for each i. Intuitively, since each time the symbolic simulation stops at the internal $match[i]$ signal, this approach avoids the OBDD size blow up problem at the *hit* signal.

The above example depicts an ideal scenario where ATPG and symbolic simulation can work together nicely to solve a problem that cannot be solved by each engine alone. Conceptually, ATPG is responsible for case-splitting of a given assertion into a set of simpler and independent sub-assertions. Each sub-assertion can then be checked by symbolic simulation separately or collectively. For example, after the ATPG processes the output OR gate, the CAM assertion will be partitioned into 8 sub-assertions where their consequents are all different. As an example, the sub-assertions for $match0[0] = 0$ and $match0[0] = 1$ are described below.

$$\begin{gathered}\text{(sub-assertion for } match[0] = 0) \\ (T[0] = T_0^{vec}) \wedge (T[1] = T_1^{vec}) \wedge (T[2] = T_2^{vec}) \wedge (T[3] = T_3^{vec}) \wedge \\ (tagin = T^{vec}) \wedge (D[0] = D_0^{vec}) \wedge \\ (D[1] = D_1^{vec}) \wedge (D[2] = D_2^{vec}) \wedge (D[3] = D_3^{vec}) \\ \Longrightarrow_{LEADTO} \\ (when(T^{vec} \neq T_0^{vec})(match[0] = 0))\end{gathered}$$

$$\begin{gathered}\text{(sub-assertion for } match[0] = 1) \\ (T[0] = T_0^{vec}) \wedge (T[1] = T_1^{vec}) \wedge (T[2] = T_2^{vec}) \wedge (T[3] = T_3^{vec}) \wedge \\ (tagin = T^{vec}) \wedge (D[0] = D_0^{vec}) \wedge \\ (D[1] = D_1^{vec}) \wedge (D[2] = D_2^{vec}) \wedge (D[3] = D_3^{vec}) \\ \Longrightarrow_{LEADTO} \\ (when(T^{vec} = T_0^{vec} \wedge T^{vec} \neq T_1^{vec} \wedge T^{vec} \neq T_2^{vec} \wedge T^{vec} \neq T_3^{vec}) \\ (match[0] = 1 \wedge dataout = D_0^{vec}))\end{gathered}$$

Notice that in both sub-assertions, the antecedents are the same as the antecedents in the original CAM assertion. This is because the antecedent specifies only initial array conditions (and inputs) as arbitrary states and hence, is not processed by the ATPG.

5 ATPG and Symbolic Simulation

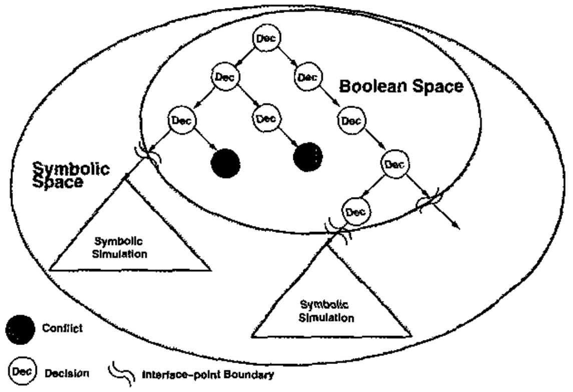


Figure 5: Combine ATPG and Symbolic Simulation

Figure 5 illustrates the overall branch-and-bound search strategy by combining ATPG and symbolic simulation. Recall that for assertion checking, we transform a given problem into an ATPG justification instance as described in Figure 3 before. Hence, the search process always starts with the ATPG justification. When enough constant values have been assigned to the circuit signals (either by implications or by ATPG decisions), the problem space is reduced. Then, symbolic simulation is called to decide if there exists a test vector to satisfy all the justification objectives under the restricted functional space. In essence, each path entering the symbolic space in Figure 5 represents a way to obtain a sub-assertion, and at the bottom of the search tree, symbolic simulation is run on each sub-assertion.

To enable the above combined ATPG/symbolic simulation search, several issues need to be resolved. They include *Constraint Modeling*, *Circuit Partitioning*, and *Sub-assertion Generation*.

Constraint Modeling:

To construct the justification instance based upon a given assertion, we need a modeling scheme to synthesize the cycle-based constraints into constraint circuitry. Since the assertion specifies constraints based upon clock cycles, within each cycle, the constraints can be thought as a combinational logic. Then, we can use latches to separate constraint circuits in different time frames (see Figure 3).

- Memory constraints are not easily recognized and synthesized correctly. To simplify the problem, memory constraints are specified implicitly using two pre-defined primitives RAM and CAM. The RAM primitive is similar to the memory primitive commonly used in commercial ATPG tools. The CAM primitive is similar to the one depicted in Figure 4 before, with more features included. One notable feature is to allow the usage of "mask bits" when performing tag comparison.

  We note that with the two memory primitives in place, the assertions would look very different from those described above. Since basic memory functionalities are implicitly encoded in the primitives, there is no need to explicitly specify the memory constraints as those assertions in the previous section. The following figure illustrates this point.

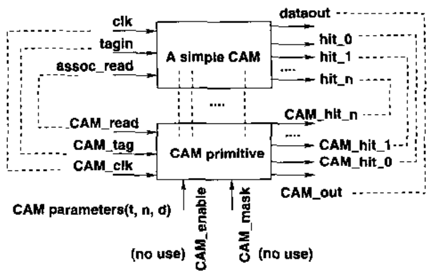


Figure 6: Specify CAM assertion using CAM primitive

  Given a simple CAM design, the CAM assertion specifies four things: 1) the input and output connections corresponding to the CAM primitives, 2) the CAM features required (e.g. CAM_enable and CAM_mask are not needed), 3) the size parameters (e.g. t,n,d specify the tag width, the number of tag entries, and data width, respectively), and 4) the memory cell corresponding relations between the design and the primitive. Then, the CAM primitive can be instantiated into an actual CAM design representing the assertion. We note that in Figure 6, the CAM primitive does not include the implementation of the final CAM *hit* signal. This should be achieved by an extra OR gate so that the OR can be partitioned into the ATPG space. Hence, the CAM primitive only includes the individual CAM_hit_*i* for all *i*.

- Each domain in the consequent of an assertion specifies the functional space for a Boolean constraint to be true. Note that outside the functional space, the value is unknown. Hence, it is intuitive to use a tri-state buffer to capture each domain specified by a *when* condition. For example, a consequent constraint can be $(whenD)(s=V)$ where both D and V are Boolean functions and s is a signal. The following figure illustrates the usage of a tri-state buffer to synthesize the constraint.

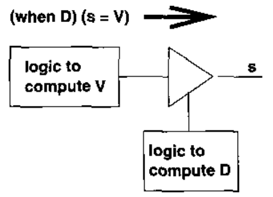


Figure 7: Synthesize domain constraint

- The domain in the antecedent of an assertion specifies the input space for which the combined ATPG/symbolic simulation should search. Since this input domain constraints are Boolean functions, they can be synthesized as combinational logic.

To summarize the discussion, Figure 8 illustrates the justification circuit instance for assertion checking.

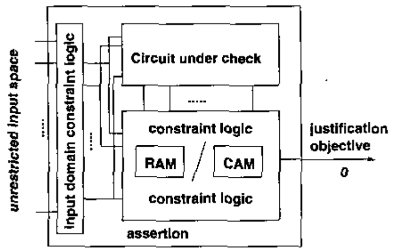


Figure 8: Illustration of Justification Circuit Instance

Circuit Partitioning:

Given a justification circuit instance, the circuit needs to be partitioned into ATPG and symbolic domains. Currently, this partitioning is static and follows the simple rules: 1) The (word-level) comparator outputs are the boundaries between the two domains. The downstream logic is given to ATPG and the upstream logic is given to symbolic simulation. 2) Logic that controls a MUX selection or an enable signal for a primitive (such as latch or tri-state buffer) belongs to ATPG domain. 3) Memory primitives belong to symbolic domain. 4) Input domain constraint logic are handled by symbolic simulation. 5) All logic that locates on the path of a word-level data bus (or address bus) going to a memory stays in the symbolic domain. 6) Finally, the output logic to the justification output is handled by ATPG.

We note that if there exists a complex Finite State Machine (FSM) in the circuit for generating control signals, then the entire FSM should be handled by the ATPG. However, for the high-performance memory designs used in our experiments, complex FSM does not really exist. Hence, currently we do not consider the handling of complex FSM. Moreover, a memory assertion often specifies functional properties that should be true within a few clock cycles. Hence, after timeframe expansion, the primary purpose of latches can be viewed simply as state holding elements in a pipeline.

Sub-assertion Generation:

Given a justification circuit instance as that in Figure 8, essentially ATPG is used to produce a simpler sub-assertion from the assertion constraint circuit and then, symbolic simulation will simulate the sub-assertion on the circuit under check to see if the assigned values by the ATPG on all signals can indeed be satisfied. We emphasize that this process is different from traditional use of ATPG for structural equivalence checking. When performing structural equivalence checking, ATPG starts from the justification objective at the primary output, and goes backward (by ATPG justification and implications) on both the golden model circuit and the circuit under check . In our approach, the analysis on the assertion constraint circuit is also done backward in order to produce a sub-assertion. However, symbolic simulation of the sub-assertion on the circuit under check proceeds forward.

It is also important to note that for assertion checking, the search space on the assertion constraint circuit is restricted by the constraints specified in the consequent. Take the simple CAM as an example. The consequent specifies that the *hit* should be a constant (1 or 0). Hence, ATPG does not need to try *hit* = 1 if *hit* = 0 is already specified (and vice verse) on the assertion circuit. From this viewpoint, solving the justification circuit instance in our approach is different from achieving the structural equivalence checking as well.

5.1 Backward Analysis of Assertion Constraint Circuit for Generating Sub-Assertions

As illustrated in Figure 5 above, ATPG will always assign enough constant values (by implications and/or decisions) so that all signals crossing the ATPG and symbolic simulation domains are holding constant logic values. At that point, backward analysis on the part of assertion constraint circuit, which belongs to the symbolic domain is done to generate a sub-assertion.

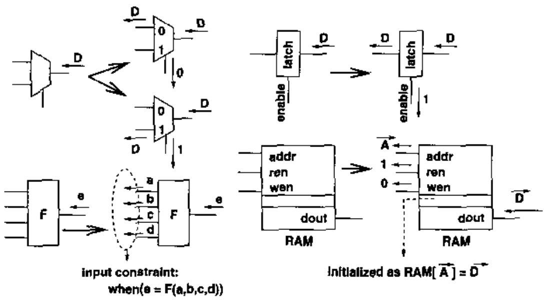


Figure 9: Illustration of Backward Analysis

Figure 9 depicts several examples to illustrate this analysis. To generate a sub-assertion based upon ATPG-assigned constant values, the goal is to produce, from the assertion circuit, symbolic inputs with input constraints satisfying both ATPG assigned values and the assertion circuit functionality. Since ATPG starts the justification process from the output, the analysis has to be done by traversing the assertion circuit backward. We describe the fundamental principles in this analysis using the four examples shown in the figure.

1. For a MUX, suppose the output is given as a ternary function D. In order to get this D, depending on the value on the selection line, either input should hold the function D. Since the selection line and its cone of logic belong to the ATPG domain, ATPG should have already assigned a value to this line before the analysis. Hence, the analysis is straightforward.
2. For other primitives such as latch, tri-state buffer with "enable" control, the analysis is straightforward as well since those controls belong to the ATPG domain.
3. For random logic as shown in case (3) in the figure, suppose a ternary function e appears at the output. In this case, four additional symbols a,b,c,d are produced. However, the values of these four symbols are constrained by $F(a,b,c,d) = e$ where F is the function implemented by the logic.
4. For a memory core instantiated from a RAM primitive (case (4)), suppose the symbolic data vector $\vec{D}$ is given. Then, the question becomes how to get $\vec{D}$ by setting up the RAM inputs and initial condition. In this case, a symbolic index $\vec{A}$ is used as both the input read address and the initialization address. Hence, the RAM is initialized by the symbolic index as $RAM[\vec{A}] = \vec{D}$. Moreover, the read enable *ren* should be high and the write enable *wen* should be low.

5.2 CAM Encoding Schemes

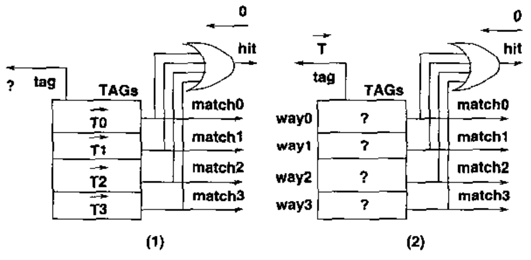


Figure 10: Illustration of CAM Problem

In backward analysis, CAM represents the most interesting case. Consider the two problems illustrated in Figure 10. At first, ATPG assigns a logic value to the *hit* signal. Suppose this value is 0. With ATPG implication, *match0*...*match4* all receive value 0. This represents a constraint on the given CAM that the input *tag* should match no tags stored in the CAM array. In backward analysis, this constraint has to be encoded into the CAM *tag* input. The problem can be formulated in two ways as depicted in the figure.

1. In (1), we assume that the CAM array is initialized with an arbitrary state represented by the four unconstrained symbolic vectors $\vec{T_0}$, $\vec{T_1}$ $\vec{T_2}$, and $\vec{T_3}$. Then, the question becomes: what values *tag* should be assigned so that it will match none of the symbolic vectors?
2. In (2), we assume that the input is arbitrary and represented by the symbolic vectors $\vec{T}$. And the question is: what values should we use to initialize the CAM so that none of the stored tags will match $\vec{T}$?

The above two formulations actually try to solve the same problem from different perspectives. In the following, we discuss their solutions.

Formulation (1):

The most straightforward method to represent the constraint that the input *tag* matches none of the stored tags in the array can be:

$$(when(\vec{T} \neq \vec{T_0}) \wedge (\vec{T} \neq \vec{T_1}) \wedge (\vec{T} \neq \vec{T_2}) \wedge (\vec{T} \neq \vec{T_3}))(\vec{T})$$

However, this encoding for *tag* essentially results in the same complexity as using the symbolic simulation alone to solve the CAM verification problem. This is because to represent the condition $(when(\vec{T} \neq \vec{T_0}) \wedge (\vec{T} \neq \vec{T_1}) \wedge (\vec{T} \neq \vec{T_2}) \wedge (\vec{T} \neq \vec{T_3}))$, we run into the same potential problem of OBDD size blow up as discussed before. Hence, we need to adopt more sophisticated encoding schemes for CAM as described below.

The fundamental idea of CAM encoding was first proposed in [14]. Here we extend the idea to solve the given problem. Consider $\vec{T_0}$ first. Assume $\vec{T_0} = [t_0, \ldots, t_{w-1}]$ where w is the tag width. In order for *tag* to mismatch with $\vec{T_0}$, *tag* should be consistent with one of the following tenary vectors: $(\neg t_0, X, \ldots, X)$, $(X, \neg t_1, X, \ldots, X)$, $\ldots$, $(X, \ldots, X, \neg t_{w-1})$.

To encode the fact that *tag* is one of the above ternary vectors, we can use a symbolic index I_0 where the width of I_0 is equal to $\lceil \log w \rceil$. Then, *tag* should be assigned the following ternary symbolic values (*tag*[0] is the most significant bit of *tag*):

$$\begin{gathered} tag[0] := (when(I_0 = 0))(\neg t_0) \\ tag[1] := (when(I_0 = 1))(\neg t_1) \\ \ldots \\ tag[w-1] := (when(I_0 = w-1))(\neg t_{w-1}) \end{gathered}$$

Then, it can be easily verified that $(when(I_0 < w))$, the comparison between *tag* and $\vec{T_0}$ will always result in a mismatch. We note that with the above scheme, all possible logic values that result in a mismatch between the *tag* and $\vec{T_0}$ are encoded.

By applying the same concept to $\vec{T_1}$, $\vec{T_2}$, and $\vec{T_3}$, we will need additional three symbolic indices I_1, I_2, I_3 each with $\lceil \log w \rceil$ wide. Then, the following shows the symbolic value assigned to *tag*'s bit 0 *tag*[0](similar assignments can be done for *tag*[1]...*tag*[$w-1$]):

$$\begin{gathered} (when(I_0 = 0) \wedge (I_1 \neq 0) \wedge (I_2 \neq 0) \wedge (I_3 \neq 0))(\neg \vec{T_0}[0]) \wedge \\ (when(I_0 \neq 0) \wedge (I_1 = 0) \wedge (I_1 \neq 0) \wedge (I_1 \neq 0))(\neg \vec{T_1}[0]) \wedge \\ (when(I_0 \neq 0) \wedge (I_1 \neq 0) \wedge (I_2 = 0) \wedge (I_3 \neq 0))(\neg \vec{T_2}[0]) \wedge \\ (when(I_0 \neq 0) \wedge (I_1 \neq 0) \wedge (I_2 \neq 0) \wedge (I_3 = 0))(\neg \vec{T_3}[0]) \wedge \\ (when(I_0 = 0) \wedge (I_1 = 0) \wedge (I_2 \neq 0) \wedge (I_3 \neq 0) \\ \wedge(\vec{T_0}[0] = \vec{T_1}[0]))(\neg \vec{T_0}[0]) \wedge \\ (when(I_0 \neq 0) \wedge (I_1 = 0) \wedge (I_2 = 0) \wedge (I_3 \neq 0) \\ \wedge(\vec{T_1}[0] = \vec{T_2}[0]))(\neg \vec{T_1}[0]) \wedge \\ \ldots \\ (when(I_0 = 0) \wedge (I_1 = 0) \wedge (I_2 = 0) \wedge (I_3 = 0) \\ \wedge(\vec{T_0}[0] = \vec{T_1}[0] = \vec{T_2}[0] = \vec{T_3}[0]))(\neg \vec{T_0}[0]) \end{gathered}$$

In the above ternary formula, the only condition that does not appear in the *when* condition is $(I_0 \neq 0) \wedge (I_1 \neq 0) \wedge (I_2 \neq 0) \wedge (I_3 \neq 0)$ which represents the case that the mismatch cannot happen at the 0th bit (most significant bit) and this is not true.

With similar ternary function encoding schemes, we can assign symbolic values to bits *tag*[1]...*tag*[$w-1$]. Then, it can be verified that after the *tag* is assigned with these symbolic values, it will not match any of the $\vec{T_0}, \vec{T_1}, \vec{T_2}, \vec{T_3}$.

Formulation (2):

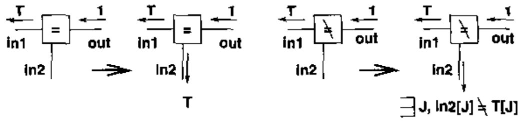
Following the same concept, we can solve the problem in (2) more easily. Given a symbolic vector $\vec{T}$, we introduce four symbolic indices J_0, J_1, J_2, J_3. Then, we use J_0 to encode the values for *way*0, J_1 for *way*1, and so on. The following repeats the encoding for *way*0:

$$\begin{gathered} way0[0] := (when(J_0 = 0))(\neg \vec{T}[0]) \\ way0[1] := (when(J_0 = 1))(\neg \vec{T}[1]) \\ \ldots \\ way0[w-1] := (when(J_0 = w-1))(\neg \vec{T}[w-1]) \end{gathered}$$

*way*1, *way*2, *way*3 can follow similar encoding schemes. Then, it is easy to check that $\vec{T}$ will not match any one of the tags stored in the CAM array [14].

CAM-hit Encoding:

The encoding scheme in formulation (2) above is simpler (use less number of symbols) and can easily be extended to represent the cases that $hit = 1$. For example, suppose that for justifying $hit = 1$, ATPG assigns 0,1,1,0 to *match*0, *match*1, *match*2, *match*3 (see Figure 10 for the definitions of these signals), respectively. Then, we introduce two symbolic indices J_0 and J_3 in this case to encode that *way*0 and *way*3 have values not equal to $\vec{T}$. For *way*1 and *way*2, we can simply assign $\vec{T}$ as the initial value. The following summarizes the backward analysis when comparison is involved.



Mask Bits:

In many CAM design, the tag comparison can be masked by given mask bits. Let *mask*[$0..w-1$] be the mask bits. Then, suppose that we want to ensure *way*0 does not match $\vec{T}$ based upon *mask*. In other word, only if *mask*[i] $= 1$ will the ith bits of *way*0 be compared to $\vec{T}[i]$. With the symbolic index J_0, the values of mask bits can be encoded simply as below.

$$\begin{gathered} mask[0] := (when(J_0 = 0))(1) \\ mask[1] := (when(J_0 = 1))(1) \\ \ldots \\ mask[w-1] := (when(J_0 = w-1))(1) \end{gathered}$$

With *mask* being encoded in this way, and *way*0 being encoded the way above, together they state the fact that $\exists J_0$ s.t. $mask[J_0] = 1 \wedge way0[J_0] \neq T[J_0]$.

5.3 Consistency Check

After backward analysis on the assertion circuit, a sub-assertion is produced. In this sub-assertion, all the constraints imposed by the assertion circuit and by the ATPG assigned constants are encoded as symbolic inputs and symbolic initial states in the arrays. Then, these symbolic constraints are simulated on the circuit under check by the symbolic simulator. The results are checked with the ATPG assigned values on the circuit. For a given signal s, suppose that ATPG assign a justification value 0 and symbolic simulation computes a ternary function S for s. Then, we check to see if S can be 0 with at least one logic value assignment to all the symbols in use. If this is possible, then assertion checking fails and the value assignment is reported as the witness vector to differentiate the assertion and the circuit. If for all signal checks they pass, then the sub-assertion checking succeeds, and the ATPG continues the search until all search space is exhausted.

6 Verification of MMU

In this section, we describe the application of our approach to verify the MMU design in Motorola high-performance microprocessors [23].

Motorola microprocessor MMU contains 64-entry, two-way set-associative, data and instruction Translation Look-aside Buffers (DTLB and ITLB) that provide support for demand-paged virtual memory address translation and variable-sized block translation. The block diagram of the instruction side MMU (IMMU) is shown in Figure 11.

MMU supports block address translation through the use of two independent instruction and data block address translation (IBAT and DBAT) arrays of four entries each. Effective addresses are compared simultaneously with all four entries in a BAT array during block translation. If the address is present in the BAT, the hit signal is 1, else it is 0. In the architecture definition, if an effective address hits in both the TLB and BAT array, the BAT translation takes priority.

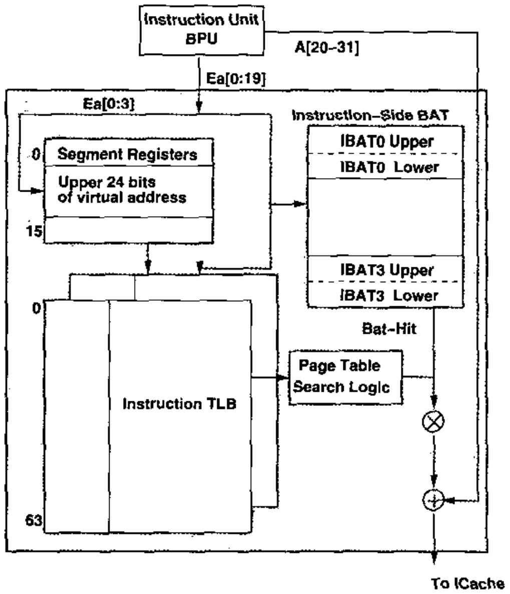


Figure 11: Block Diagram of IMMU [23]

The implementations of the instruction side MMU (IMMU) and data side MMU (DMMU) are similar. Hence, in our work we only focus on IMMU. One particular MMU functionality to which we pay more attention is the address translation that involves both BAT and TLB. That is, the effectiveness address ea misses all four entries in BAT, and the TLB and segment register file (SEG) are responsible to produce the physical address. This assertion is interesting because it enforces a dependency of TLB on BAT and hence, results in a system-level assertion. Another interesting case is to verify BAT alone. Due to the content addressable nature of the BAT, verification of BAT using symbolic simulation was not very efficient before. Hence, we are also interested in knowing how effective the ATPG and symbolic simulation combined strategy will be for the BAT.

6.1 IBAT Unit

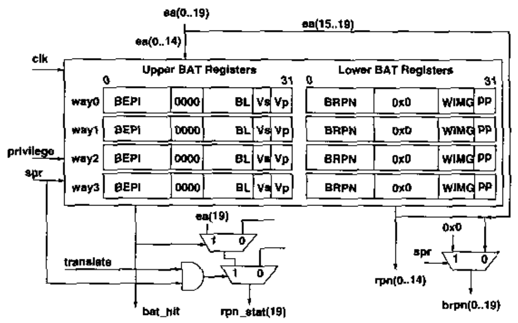


Figure 12: Simplified BAT Organization

Figure 12 illustrates the BAT organization. The BAT is organized as a 4-way CAM. In each way, the registers are organized as upper registers (32 bits) and lower registers (32 bits). In the non-SPR mode where $spr = 0$, the BAT translates the i ($9 \leq i \leq 15$) most significant bits of the effective address ea into the physical address via the CAM associative read operation. The remaining ea bits pass unchanged.

In BAT address translation, the incoming effective address $ea(0..14)$ is compared to the 15-bit Block Effective Page Index ($BEPI$) entries. Each entry comparison is masked by the 11-bit Block Length (BL) on the 5th to 15th bits of ea (and $BEPI$). In the functional mode, the legal combinations of BL are $00\cdots0, 00\cdots01, 00\cdots011, \ldots, 011\cdots1$, and $11\cdots1$. A 0's indicates that the bit should be compared. If BL is all 1's, then only the most significant 4 bits are compared.

Each tag entry contains two "valid" bits (Vs, Vp) to indicate if $BEPI$ is valid. Which valid bit is used is controlled by the incoming signal $privilege$. If an entry valid bit is 0, then the entry comparison fails by default.

When there is a match in $BEPI$, the corresponding 15-bit Block Real Page Number $BRPN$ is sent out as the upper 15-bit of the physical address $brpn(0..19)$.

6.2 BAT Results

We focus on the assertion where ea does not match any of the $BEPI$ entries (i.e. $bat\_hit = 0$). From past experiences, this assertion is the most difficult one to be verified by OBDD-based symbolic simulation [2]. In the experiments, we consider various cases as explained below.

SS-alone For these experiments, we discuss two cases, one with a non-optimized variable ordering and the other with a carefully optimized manual ordering.

ATPG/SS For these experiments, we also study two cases, one using the encoding scheme described in **Formulation (1)** in Section 5.2 and the other using the encoding scheme described in **Formulation (2)**.

6.2.1 SS-alone

symbolic bits	1	2	3	4	5	6	7	> 7
time (sec)	1.4	1.8	2.3	46.8	53.3	86.8	177.1	abort

Table 1: BAT Results with Non-Optimized Ordering

In the initial experiments, for debugging the program we often assign less number of symbolic bits to both *BEPI* and *ea*. For example, to use only 3 symbolic bits, we can have $BEPI[0..14] = [B_0, B_1, B_2, 1, 1, \ldots 1]$ and $ea[0..14] = [ea_0, ea_1, ea_2, 1, 1, \ldots, 1]$. In this way, the symbolic simulation complexity can be reduced.

Without any manual effort to adjust the variable ordering, symbolic simulation cannot finish the BAT miss assertion even for the case where only 3 out of the 15 bits in *BEPI* and *ea* are assigned with symbolic values. With the manual effort trying to interleave the symbolic vectors, symbolic simulation can handle up to 7 bits of symbolic values. Table 1 shows the run times for using from 1 to 7 symbolic bits in *ea* and *BEPI*. If more than 7 symbolic bits are used, then the run time would take too long. OBDD re-ordering would consume too much time during the symbolic simulation and we aborted the run after waiting for some time (say, 5-10 minutes). All our experiments were run on a Pentium 4 1.5G machine running Linux Mandrake 2.4.8-26mdk with 512M memory.

symbolic bits	3-bit	6-bit	10-bit	11-bit	15-bit*
time (sec)	8.6	11.9	74.2	127.1	377.3
total OBDD nodes	4531	8485	12271	14991	28395
max OBDD nodes	133886	141924	186173	199650	392544

*No constant logic values are used

Table 2: BAT Results with Manually Optimized Ordering

We manually tuned the OBDD ordering and repeated the experiments. After enough effort, we were able to find a good ordering that allows symbolic simulation to run. Table 2 shows the results. In this sequence of experiments, we were able to complete the assertion check without using any constant logic values to simplify the assertion. By comparing the results in Table 1 and in Table 2, we observe that variable ordering significantly impacts the performance of symbolic simulation.

For the OBDD sizes, we show two types of data: the total number of OBDD nodes at the end of symbolic simulation (total OBDD nodes), and the maximum number of OBDD nodes during the symbolic simulation (max OBDD nodes). The total number of OBDD nodes depends on the design functionality and the variable ordering at the end (after dynamic ordering). The maximum number of OBDD nodes depends on the implementation and the initial ordering given.

6.2.2 ATPG/SS

Table 3 presents the results by using ATPG and symbolic simulation combined. Here we compare two different CAM encoding schemes from **Formulations (1)** and **(2)**. We note that the BAT is a 4-way design. However, the results shown in the first row were obtained by disabling 2 entries in the BAT and making it only 2-way. Our intention was to show how sensitive the encoding scheme "(1)" was to the number of BAT entries.

	Time (sec)	OBDD nodes total	max
Formulation (1)*, 2-way	22.1	1770	589824
Formulation (1)*, 4-way	273.9	291259	692091
Formulation (2)*, 4-way	0.2	377	< 500

*indicate which CAM encoding is used

Table 3: BAT Results with ATPG/SS Combined

As it can be observed, the encoding scheme "(2)" is much more efficient. The performance of the encoding scheme "(1)" is very sensitive to the number of BAT entries and hence, the scheme might not scale well. We note that the encoding scheme "(1)" uses almost 4 times more symbolic values than the encoding scheme "(2)." Hence, intuitively the encoding scheme "(2)" should be better. For results in the row Formulation (2) in Table 3, we emphasize that no manual effort is required to adjust the variable ordering.

6.3 TLB Results

	Time (sec)	total OBDD nodes
SS-alone*	2.8	10219
ATPG/SS	3.1	12149

* with manually optimized ordering

Table 4: TLB Results for $tlb\_hit0 = 1, tlb\_hit1 = 0$

The TLB is organized as a 2-way 64-entry array, addressable by a TLB index *tindex*[0..5] (Figure 11). For address translation, the two 35-bit tags stored in the entry pointed by *tindex* are compared to the effective address *ea* and data from the segment register. When way 0 entry matches, the address is computed from the data stored in way 0 and $tlb\_hit0 = 1$ (and vice verse).

Table 4 shows results based upon the assertion for $tlb\_hit0 = 1$ and $tlb\_hit1 = 0$. We emphasize that for symbolic simulation alone, if no manual effort is involved to adjust the variable ordering, the simulation would abort when more than 19 symbolic bits are used. With 18 symbolic bits, the symbolic simulation took 124.3 seconds to complete and the maximum number of OBDD nodes was about 4.5M, and with 20 symbolic bits, the maximum number of OBDD nodes exceeds 22M and the run aborted. However, for ATPG/symbolic simulation combined, no special ordering is required. In ATPG/SS, we adopt the "CAM-hit" encoding scheme as described in Section 5.2.

6.4 MMU Results

	Time (sec)	total OBDD nodes
ATPG/SS	8.3	12738

*SS-alone aborts in this case

Table 5: MMU Results for ($bat\_hit = 0, tlb\_hit = 1$)

Table 5 demonstrates the results by using ATPG and symbolic simulation combined to verify the address translation assertion defined on the whole MMU. In this assertion, the outputs of the TLB depend on the comparison results in the BAT. Symbolic simulation alone was not able to finish the run within a reasonable time. In ATPG/SS, the Formulation (2) CAM encoding is used. The total number of OBDD nodes was the maximum among all sub-assertion runs. The combined strategy can finish the checking of the assertion in less than 9 seconds!

7 Conclusion

In this paper, we present a novel approach to combine ATPG and symbolic simulation for efficient system-level validation of embedded memories. Our approach allows ATPG justification to partition a given assertion into a sequence of simpler sub-assertions to be checked by the ternary symbolic simulation. We compare the new method to symbolic simulation, and demonstrate the effectiveness of the new method through experiments on Motorola microprocessor MMU. Our experience indicates that without combining with the ATPG, the performance of the OBDD-based symbolic simulation is sensitive to the initial variable ordering given. In many cases, manual effort is required to identify a good variable ordering for symbolic simulation to run. With our ATPG and symbolic simulation combined approach, the performance is less sensitive to the initial variable ordering. ATPG and symbolic simulation together is able to verify a system-level assertion that was not possible to verify by using just symbolic simulation alone. In our future work, we will apply the new methods to other memory designs as well as to larger memory systems.

Acknowledgement: Authors would like to thank Dr. Manish Pandey and Dr. Ric C.-Y. Huang at Verplex Systems, Inc. for their inspiring discussion on the project.

References

[1] N. Ganguly and M. S. Abadir and M. Pandey. PowerPC Array Verification Methodology Using Formal Verification Techniques. In *Proc. IEEE Int. Test Conference (ITC)*, pages 4 857–864, Washington, DC, Oct. 1998. IEEE Computer Society Press.

[2] Li-C. Wang and M.S. Abadir. Experience in Validation of PowerPC Microprocessor Embedded Arrays. *Journal of Electronic Testing: Theory and Applications (JETTA)*, 15:191–205, 1999.

[3] R.E. Bryant. Algorithmic aspects of symbolic switch network analysis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(4):618–633, July 1987.

[4] R.E. Bryant. Boolean analysis of MOS circuits. *IEEE Trans. on Computer-Aided Design*, CAD-6(4):634–649, July 1987.

[5] R.E. Bryant. A methodology for hardware verification based on logic simulation. Technical Report CMU-CS-87-128, Computer Science Department, Carnegie Mellon University, Pittsburgh, June 1987.

[6] D.L. Beatty, R.E. Bryant, and C.-J.H. Seger. Synchronous circuit verification by symbolic simulation: An illustration. In W.J. Dally, editor, *Proc. of the sixth MIT Conference on Advanced Research in VLSI*, pages 98–112, Cambridge, 1990. MIT Press.

[7] R.E. Bryant. Formal verification of memory-circuits by symbolic-logic simulation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 10(1):94–102, 1991.

[8] R.E. Bryant and C.-J.H. Seger. Formal verification of digital circuits using symbolic ternary system models. In E.M. Clarke and R.P. Kurshan, editors, *Proc. of the Workshop on Computer-Aided Verification (CAV90)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, New York, 1991. American Mathematical Society, Springer-Verlag.

[9] R.E. Bryant, D.L. Beatty, and C.-J.H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *28th ACM/IEEE Design Automation Conference*, pages 397–402, 1991.

[10] C.-J.H. Seger and R.E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in Systems Design*, 6:147–189, Mars 1995.

[11] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[12] M. Pandey, R. Raimi, D. Beatty, and R.E. Bryant. Formal verification of powerpc^{tm} arrays using symbolic trajectory evaluation. In *33rd ACM/IEEE Design Automation Conference (DAC)*, Las Vegas, NV, 1996.

[13] Narayanan Krishnamurthy et. al. Validating PowerPC Microprocessor Custom Memories. In *IEEE Design and Test of Computers*, Oct-Dec 2000, pages 61–76.

[14] M. Pandey, R. Raimi, R.E. Bryant, and M.S. Abadir. Formal verification of content addressable memories using symbolic trajectory evaluation. In *34th ACM/IEEE Design Automation Conference (DAC)*, Las Vegas, NV, 1997.

[15] 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, November 1993.

[16] Li-C. Wang, Magdy S. Abadir, and N. Krishnamurthy. Automatic Generation of Assertions for Formal Verification of PowerPC Microprocessor Arrays Using Symbolic Trajectory Simulation. In *35th ACM Design Automation Conference*, 1998.

[17] Chris Wilson, David L. Dill, and Randal E. Bryant. Symbolic Simulation with Approximate Values, In *Proc. of the Third International Conference on Formal Methods in Computer-Aided Design*, Austin, Texas, November 2000

[18] Chris Wilson and David L. Dill. Reliable Verification using Symbolic Simulation with Scalar Values, *37th Design Automation Conference*, June 2000

[19] Chris Wilson. *Symbolic Simulation Using Automatic Abstraction of Internal Node Values*, PhD Thesis, Stanford University, October, 2001

[20] C.-Y. Huang and K.-T. Cheng. Assertion Checking by Combined Word-level ATPG and Modular Arithmetic Constraint-Solving Techniques. In *37th ACM/IEEE Design Automation Conference (DAC)*, June 2000.

[21] S. Hazelhurst and C.-J.H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDD's. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 14(4):413–422, April 1995.

[22] M. Aagaard and C.-J.H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *ACM/IEEE Int. Conference on Computer-Aided Design*, pages 7–10, November 1995.

[23] *PowerPC^{TM} Microprocessor Family: The Programming Environments*, Motorola Inc., 1994.