Design Automation for Embedded Systems, 8, 173–188, 2003.
© 2003 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Enhanced Symbolic Simulation for Functional Verification of Embedded Array Systems

LI-C WANG, TAO FENG, AND KWANG-TING (TIM) CHENG
Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, USA

MAGDY S. ABADIR
ASP Advanced Tools and Methodologies, Motorola Inc., Austin, TX, USA

MANISH PANDEY
Verplex Systems, Inc., Milpitas, CA, USA

Abstract. Symbolic simulation is an effective approach for verifying individual array blocks. This paper presents two methods to enhance the capacity of symbolic simulation for handling large and complex embedded array systems. The first method combines an ATPG decision procedure with symbolic simulation. By developing a scheme that enables the ATPG to work effectively with a symbolic simulator, the run-time OBDD sizes can be limited. In the second method, we propose a "dual-rail" symbolic simulator where a given design is partitioned implicitly into control and datapath domains. Symbolic simulation is carried out simultaneously on both domains. We demonstrate and compare the effectiveness of both methods based on verification of the Memory Management Unit (MMU) in Motorola high-performance microprocessors.

Keywords: ATPG, array verification, Symbolic simulation.

1. Introduction

Embedded arrays are important components in many high-performance digital ICs. The speed of these on-chip memory components is critical to the overall performance of a chip. They are often custom-designed at transistor level. A typical array system contains multiple array blocks, and interactions among these blocks can be complex and hard to verify.

The size, complexity, and sequential nature of embedded arrays make their test and verification very challenging. Because they are custom designs, correct modeling can be time-consuming and error-prone. For both functional verification and structural equivalence checking of these arrays, researchers [1], [2] demonstrated that Symbolic Trajectory Evaluation (STE) could be an effective approach. However, past results were shown only based on verification of individual array blocks [2]–[6].

In STE, symbolic simulation is the underlying engine [7]–[9] utilizing *Ordered Boolean Decision Diagrams* (OBDDs) [10] to manipulate ternary logic functions. When verifying an operation that involves interactions among multiple array blocks, the OBDD sizes can often blow up. Hence, in this paper, our goal is to enhance the capacity of symbolic simulation so that multiple array blocks can be verified together as a single system. We propose two approaches to enhance the capacity of symbolic simulation. We compare the strengths and weaknesses of these two approaches.

Our first approach incorporates an ATPG decision procedure into the symbolic simulation framework [11]. In this approach, symbolic simulation is partitioned into separate and simpler subprocesses by ATPG assigning constant values to a set of selected control signals. Because the control space is partitioned, *symbolic encoding* can be applied to optimize symbolic simulation on the datapath.

Our second approach is different from the first approach in terms of three aspects [12]: (1) instead of explicit partitioning a design into control and datapath, the approach adopts a partition scheme implicitly in the symbolic simulation, (2) instead of using a decision procedure to handle the control part, it uses symbolic simulation on both the control and the datapath, (3) instead of using symbolic encoding to optimize symbolic simulation on the datapath, it adopts a *node collapsing* method that achieves the same efficiency. We call the second approach *dual-rail symbolic simulation*.

To demonstrate the effectiveness of these two approaches, we apply them to verify the Memory Management Unit (MMU) in Motorola high-performance processors. Verifying the entire MMU with symbolic simulation was not possible before due to the OBDD size blow-up when an ordinary symbolic simulator is used in STE.

The idea of partitioning functional space for symbolic simulation to handle large and complex designs is not new. For example, the authors in [13], [14] proposed using a combined SAT and OBDD-based symbolic simulation approach to avoid OBDD blow-up. The authors in [15] proposed a hybrid approach that was able to verify a 64-bit multiplier by verifying individual adders using STE and then composing the results to show that the adders were properly connected. The authors in [16] used the hybrid approach to verify a radix-eight, pipelined, IEEE double-precision floating point multiplier. The authors in [17] used a functional space decomposition approach to verify an arithmetic circuit. With good understanding of the design, the authors manually decomposed the verification task into a number of sub-cases, and verified each sub-case independently. Since each sub-case involves only a part of the design functionality, input constraints may be required to process each particular sub-case. *Encoding* techniques can be applied to enforce such constraints. Our first method is similar to that in [17]. The key difference is that we use an ATPG decision procedure for the decomposition of the functional space. Moreover, our encoding techniques are different. Our second method tried to achieve the same effectiveness as the first method, without using the ATPG.

2. STE and Symbolic Simulation

In STE, specifications are given as *trajectory assertions* of the form $Antecedent \Rightarrow_{LEADTO} Consequent$ where both Antecedent and Consequent consist of *trajectory formulae*. Assertions specify a set of design properties in a restricted temporal logic form, and 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 have 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 [18]. 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, $node_i$ has the value V. Otherwise, the value is unknown. Such a specification usually has two semantic meanings depending on when it is placed. If it appears in the antecedent, then for the 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 $node_i$ should only be checked under the restricted domain D. The value of $node_i$ under the domain D' is ignored.

2.1. STE as Cycle-Based Constraint Solving Process

In our work, we assume that symbolic simulation is cycle-based, and the assertions for an array unit (which may contain multiple array blocks) are given by the user. The goal is to check that the RTL model satisfies the given assertions. We adopt a cycle-based constraint-solving view for the STE.

Conceptually, trajectory formulae, specified in both antecedent and consequent, are *constraints* on circuit signals. These constraints are ternary logic functions. In cycle-based simulation, 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 the constraints imposed by the antecedent cannot be satisfied, this represents an "over-constrained" situation (meaning that the input constraints are inconsistent) [18]. When the constraints imposed by the consequent cannot be satisfied, then the design fails to satisfy the assertion.

The problem of simultaneously satisfying all constraints can be modeled as an ATPG justification problem instance illustrated in Figure 1 [11]. In such a formulation, constraints are synthesized into constraint circuitry. The task of the 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.

3. The Target Example—MMU

Our work targets on the MMU design in Motorola high-performance microprocessors [19]. The MMU design is illustrated in Figure 2. The MMU contains a 64-entry, two-way set-associative, Translation Look-aside Buffers (TLB) that provides support for demand-paged virtual memory address translation. The MMU also supports variable-sized block address translation through the use of the Block Address Translation (BAT) array. The BAT contains four entries, and up to 15 bits of the effective address (EA) are compared simultaneously with all four upper BAT entries during the address translation operation. If the effective address matches any of the upper entries in the BAT, the output *bathit* signal is 1, else it is 0. If there is a hit, the corresponding lower entry will be the address output. In the architecture definition, if an effective address hits in both the TLB and the BAT, the BAT translation takes priority.

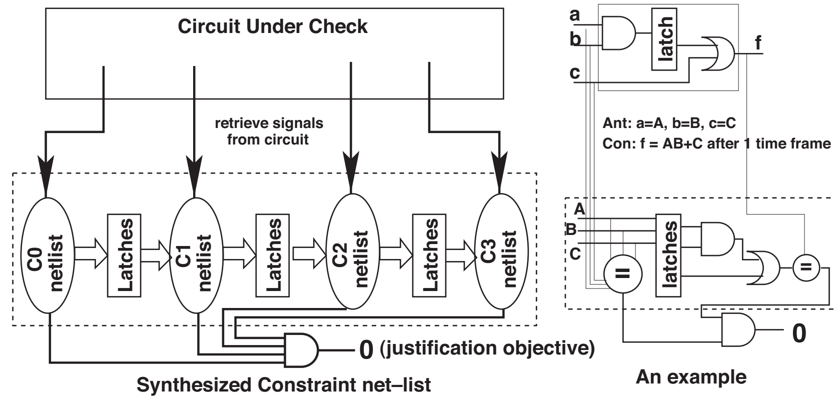

Figure 1. Constraint solving as ATPG justification.

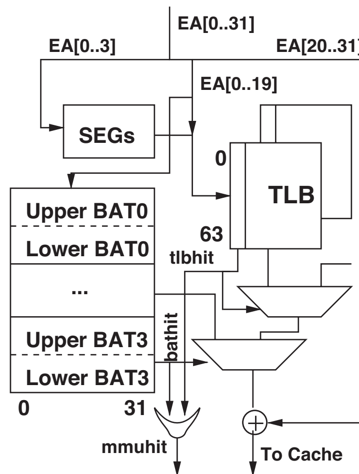

Figure 2. Block diagram of MMU.

One particular MMU functionality to which we pay more attention is the *translation* operation that involves both the BAT and the TLB. That is, the effectiveness address EA misses all four entries in BAT, and the TLB and the Segment Register (SEG) are responsible to produce the physical address. Another interesting case is to verify the BAT alone. Figure 3 illustrates the detailed BAT organization.

The BAT is organized as a four-way Content Addressable Memory (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 CAM associative read operation. The remaining ea bits pass unchanged.

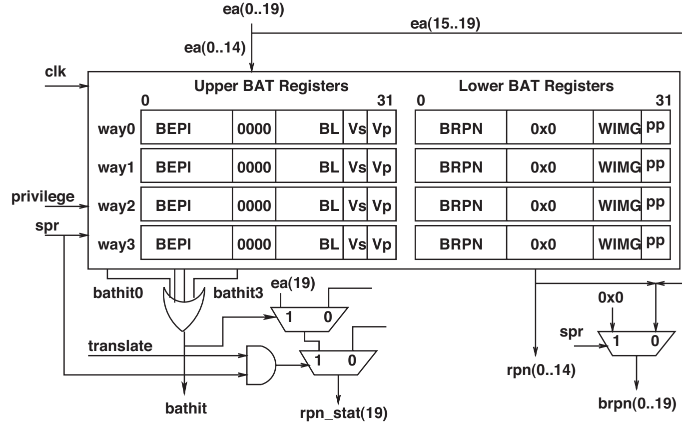

Figure 3. Simplified BAT organization.

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...0, 00...01, 00...011,..., 011...1, and 11...1. 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).

4. Method I: Partitioning The Verification Process by ATPG

Given a BAT assertion, an intuitive point to separate the control from the datapath would be the OR gate for generating the *bathit* signal. The function computed by the OR gate includes the function $(BEPI_0 = ea) \mid (BEPI_1 = ea) \mid (BEPI_2 = ea) \mid (BEP3_0 = ea)$ where $BEPI_i$ denotes the *BEPI* bits in way *i* of the BAT. The "=" denotes the equality checking by a 15-bit comparator, and the "|" denotes the logic OR. Without a good initial ordering, the OBDD size for the *bathit* function can easily blow up.

Figure 4 illustrates a partition of the BAT into control (ATPG) and datapath (symbolic simulation) domains. The control is handled by ATPG, and the datapath is handled by symbolic simulation. The BAT assertion is synthesized into a constraint circuit using a CAM primitive [11].

Suppose that in the assertion, we have $camhit = 0$. To verify that the BAT behaves the same as the four-way CAM given that $camhit = 0$, we proceed with the following

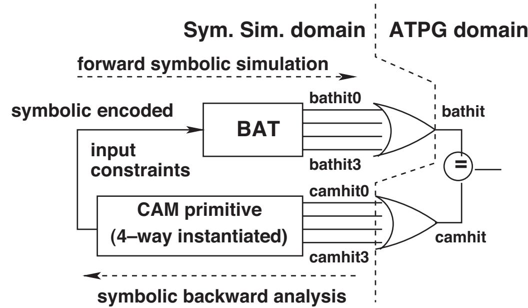

Figure 4. Combine ATPG and symbolic simulation.

three steps. (1) The ATPG assigns 0's to all individual *camhit* signals (*camhit*0–*camhit*3), (2) a *backward symbolic analysis* on the CAM produces symbolic constraints based on the ATPG-assigned values, (3) the symbolic constraints are simulated on the BAT to check if indeed, a 0's is obtained at the *bathit* signal.

Suppose that $camhit = 1$ in the assertion. In the first step, the ATPG would enumerate all possible input possibilities at the OR gate. As a result, the above three steps will be repeated 15 times. Hence, for complete verification of the BAT, the ATPG will generate 16 sub-cases (or sub-assertions).

Note that the ATPG processes the circuit *backward* while the symbolic simulation simulates the circuit *forward*. In order for these two engines to work together, a special backward analysis process is required to produce the required constraints for symbolic simulation. We call this process the *symbolic backward analysis*.

4.1. Symbolic Backward Analysis with the CAM

The symbolic backward analysis on the CAM primitive can be illustrated by Figure 5. Assume that the *hit* signal is 0, the symbolic backward analysis is to produce the symbolic constraints that satisfy $hit = 0$. These constraints can be produced in terms of the CAM's input tag (case 1) or the CAM's initial states (case 2). In case 1, the initial states in the CAM have been set with four symbolic vectors $\vec{T}_0 \ldots \vec{T}_3$. We need to constrain the symbolic input *tag* such that it will not match any of the four symbolic vectors. In case 2, the input symbolic vector is given as $\vec{T}$. We need to constrain the initial CAM states such that $\vec{T}$ will match none of the tags stored in the CAM. In the following, we discuss how to encode the constraint in case 2. The encoding scheme for case 1 is more complicated [11]. For the purpose of verifying the BAT, both types of constraints can be used. Here, we present only the solution to the case 2.

Given the symbolic vector $\vec{T}$, we introduce four symbolic indices J_0, J_1, J_2, J_3. Then, we use J_i to encode the initial state in *wayi*. The following represents the encoding for the symbolic content that should be stored in *way*0. "*way*0[$w-1$]" and "$\vec{T}[w-1]$" denote the most significant bits of the register in *way*0 and the input

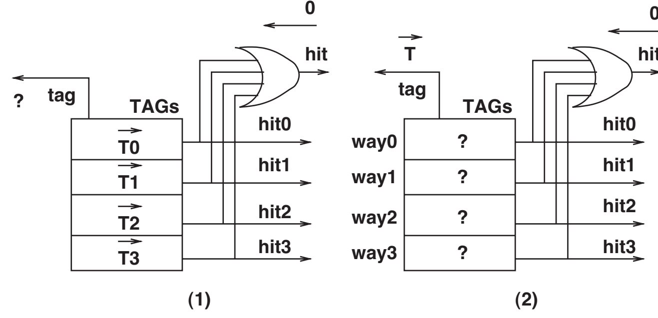

Figure 5. Illustration of backward analysis with the CAM.

tag $\overrightarrow{T}$, respectively.

$$way0[0] := (when(J_0 = 0))(\neg \overrightarrow{T}[0])$$
$$way0[1] := (when(J_0 = 1))(\neg \overrightarrow{T}[1])$$
$$\cdots$$
$$way0[w-1] := (when(J_0 = w-1))(\neg \overrightarrow{T}[w-1])$$

*way*1, *way*2, *way*3 can follow similar encoding schemes. Then, it is easy to check that $\overrightarrow{T}$ will match none of the tags stored in the CAM [4].

The above encoding scheme can be extended easily to represent the cases that camhit = 1. For example, suppose that for justifying *camhit* = 1, ATPG assigns 0, 1, 1, 0 to *camhit*0–*camhit*3. Then, we introduce two symbolic indices J_0 and J_3 to encode that the initial states in *way*0 and *way*3 do not match $\overrightarrow{T}$. For *way*1 and *way*2, we can simply assign $\overrightarrow{T}$ as their initial values.

4.2. Combining Symbolic Simulation with ATPG

To implement the combined ATPG and symbolic simulation method described above, we need to resolve the following three issues: *constraint modeling and synthesis, circuit partitioning,* and *symbolic backward analysis* in general. We summarize the key ideas below [11].

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 (as that illustrated in Figure 1 before). The key in constraint modeling is to utilize two predefined 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 used in the BAT example before. In our verification methodology, we enforce that user specifies an assertion with these primitives.

Given a justification circuit instance, the constraint circuit needs to be partitioned into ATPG and symbolic domains. The partitioning is not necessary for the circuit under verification. The partitioning is static and follows the simple rules: (1) the (word-level) comparator outputs are the boundaries between the two domains (as that shown in the BAT example), (2) logic that controls a MUX selection or an enable signal for a primitive (such as latch or tri-state buffer) belongs to the 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 array designs, complex FSM usually does not exist. Hence, currently we do not consider the handling of a complex FSM.

In backward analysis, we need to process the part of constraint circuit, which is partitioned into the symbolic domain. Figure 6 depicts several examples to illustrate this analysis. In each case, a symbolic value (or vector) is given at the output of a primitive. Input or initial state constraints are derived based on the functionality of the primitive. Figure 7 illustrates the backward analysis on a word-level comparator.

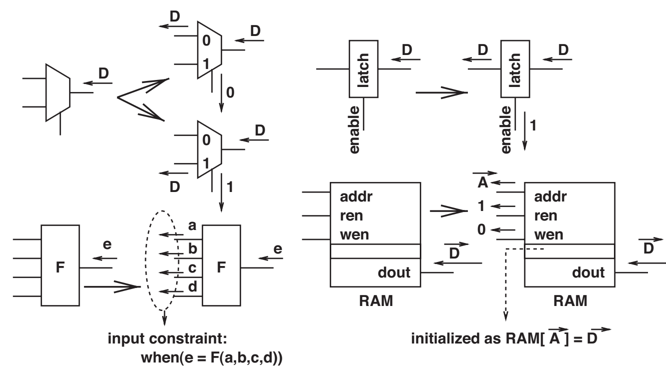

Figure 6. Illustration of symbolic backward analysis.

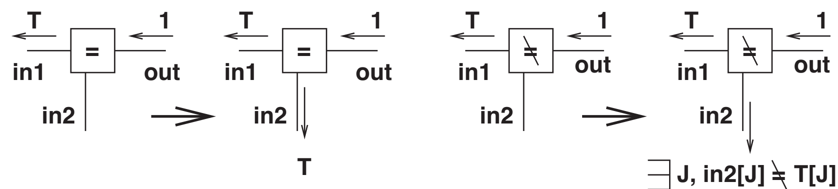

Figure 7. Backward analysis on a word-level comparator.

5. Experimental Results

All our experiments were run on a Pentium 4 1.5 G machine running Linux Mandrake 2.4.8–26 mdk with 512 M memory. We first focus on the BAT assertion where *ea* matches none of the *BEPI* entries (i.e., $bathit = 0$).

SS-alone We use a Motorola in-house symbolic simulator that was modified from *Voss* [18]. We experimented with two cases: one with the variable ordering same as the design's primary input ordering, and the other with a manually-optimized input variable ordering.

ATPG/SS The combined ATPG and symbolic simulation method was implemented as a separate tool.

Without manual effort to adjust the initial variable ordering, symbolic simulation cannot finish the BAT miss assertion where more than seven out of the 15 bits in *BEPI* and *ea* are assigned with symbolic values (other bits are assigned with 0 or 1). OBDD reordering would consume too much time during the symbolic simulation and we aborted the run after waiting for 10 min.

With manual effort, we were able to find a good ordering that allows symbolic simulation (SS) to run. Table 1 shows the results. For example, in the 3-bit experiment, the three most significant bits of *BEPI* and *ea* were given with symbolic values. Other bits were fixed with the constants 0 and 1. 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. The last column in Table 1 shows the results on the combined ATPG and symbolic simulation method.

Table 2 shows results on the TLB and the MMU. For TLB, the assertion is with the constraints $tlbhit0 = 1$ and $tlbhit1 = 0$, i.e., a hit on *way*0. The TLB is organized as a two-way (*way*0, *way*1) 64-entry array, addressable by a TLB index *tindex*[0..5]. For address translation, the two 35-bit tags stored in the entry pointed by the *tindex*, are compared to the effective address *ea* and the content of the segment register. When the tag in *way*0 matches, the address is computed from the data stored in *way*0 and $tlbhit0 = 1$ (and vice versa). For symbolic simulation alone, if no manual effort is

Table 1. Results on The BAT Miss Assertion

Symbolic Bits	3-Bit	6-Bit	10-Bit	11-Bit	15-Bit	ATPG/SS
Time (s)	8.6	11.9	74.2	127.1	377.3	0.2
Total OBDD nodes	4531	8485	12,271	14,991	28,395	377
Max OBDD nodes	133,886	141,924	186,173	199,650	392,544	496

Table 2. Results on TLB (Hit on *way*0) and MMU (BAT Miss, TLB Hit)

	TLB		MMU	
	Time (s)	**Total OBDD nodes**	**Time (s)**	**Total OBDD nodes**
SS-alone[1]	2.8	10,219	Abort	Abort
ATPG/SS	3.1	12,149	8.3	12,738

1. With manually optimized ordering

involved to adjust the variable ordering, the simulation would abort when more than 19 symbolic bits in each of the two TLB tags are used. With 18 symbolic bits, the symbolic simulation took 124.3 s to complete and the maximum number of OBDD nodes was about 4.5 M. With 20 symbolic bits, the maximum number of OBDD nodes exceeded 22 M and the run aborted. However, for ATPG/SS, no manual ordering is required. The MMU assertion is the address translation assertion where the BAT generates a miss and the TLB generates a hit. The combined ATPG/SS strategy can finish the checking of the assertion in less than 9 s.

6. Method II: Dual-Rail Symbolic Simulation

From the above experiments, we can see that the combined ATPG and symbolic simulation method can be much more efficient than the symbolic simulation alone. However, the combined method presents two difficulties in implementation.

1. Since ATPG justification is a backward process and symbolic simulation is a forward process, the *symbolic backward analysis* is required to produce the symbolic constraints based upon ATPG-assigned constant values on the constraint circuit. Symbolic backward analysis utilizes predefined primitives for simplification. This requires that we implement a separate interpreter for these primitives in order to conduct the analysis.

2. Using the CAM-based symbolic encoding is essential for obtaining the high efficiency. This means that the knowledge of the encoding should be built in with the CAM primitive and be processed by the backward analysis tool.

To avoid the complexity of backward analysis and the CAM encoding scheme, our goal in this section is to provide an alternative method that achieves the same efficiency as the first method and that is more natural to implement. In our second method, we utilize symbolic simulation to replace the ATPG. The result is a *dual-rail* symbolic simulation scheme. In this method, we replace the CAM encoding with *OBDD node collapsing* based on identifying word-level equality functions.

6.1. Replacing CAM Encoding with OBDD Node Collapsing

Section 4.1 discusses the encoding of constraints for a particular hit/miss combination. Suppose that we want to encode all possible initial states in a four-way CAM based on a given input tag. Notice that for a CAM, there can be only two outcomes from each tag entry comparison: 1 (match) and 0 (mismatch). Hence, we can introduce a new variable m_i to indicate if the tag in entry i, $T[i]$, matches the incoming tag $\overrightarrow{T}$ or not. Then, the comparison between $\overrightarrow{T}$ and $T[i]$ will always result in two possible outcomes: match (m_i) and mismatch (m_i'). Assume that $\overrightarrow{T} = [t_0, \ldots, t_{w-1}]$, where w is the tag width. The following explains the symbolic encoding for $T[i]$, $i = 0, 1, 2, 3$.

$$\begin{aligned} T[i][0] &:= ((when(m_i' \wedge (I_i = 0)))(\neg t_0)) \wedge ((when(m_i)(t_0)) \\ T[i][1] &:= ((when(m_i' \wedge (I_i = 1)))(\neg t_1)) \wedge ((when(m_i)(t_1)) \\ &\ldots \\ T[i][w-1] &:= ((when(m_i' \wedge (I_i = w-1)))(\neg t_{w-1})) \wedge ((when(m_i)(t_{w-1})) \end{aligned} \tag{1}$$

I_i is $\lceil \log w \rceil$ wide to indicate which bit has the mismatch. With this encoding, symbolic simulation on the CAM will result in m_0, m_1, m_2, m_3 at the four hit outputs. Then, the final CAM hit output will be the OR of the four symbols, m_0, m_1, m_2, m_3. If we replace each m_i with a constant value 0 or 1, then the above encoding scheme can be reduced to one of the encoding schemes discussed in Section 4.1.

The CAM encoding scheme just described is efficient because essentially, it *re-starts* the computation at each comparator output by introducing new variables (m_i). Effectively, it breaks the symbolic simulation into two separate runs, one before the comparator output and the other after the comparator.

In our second approach, the simulator introduces a new symbolic variable whenever an equality function such as "$A = B$" of two symbolic input vectors A, B, is identified during the course of symbolic simulation. We call this method *the OBDD node collapsing*. These equality functions can be extracted from the assertions specified by the user. For example, suppose in the assertions, symbolic vectors of equal lengths are defined (such as $\overrightarrow{T}$, $\overrightarrow{T}_0$). During the course of symbolic simulation, the simulator will check if the computed OBDD of a signal *contains* the equality functions "$\overrightarrow{T} = \overrightarrow{T}_0$." The equality function can be the output result from a word-level comparator primitive *COMP(tag*, *T*[0]). However, the comparator primitive may not be well defined in a real design and moreover, $\overrightarrow{T}$ and $\overrightarrow{T}_0$ may have been combined with other control symbols before reaching the comparator. Hence, the simulator needs to constantly monitor the computed results to identify that the desired equality function has been computed. Various heuristics were used to minimize the need for constantly monitoring the computed results [12]. Figure 8 shows the flow of the OBDD node collapsing method.

The OBDD node collapsing also intends to *restart* the symbolic simulation internally at each word-level comparator output. However, it is fundamentally different from the CAM encoding. The process of CAM encoding is *memory-less*: At each comparator output, all information beforehand (ex. $\overrightarrow{T}$) is lost. The simulation result

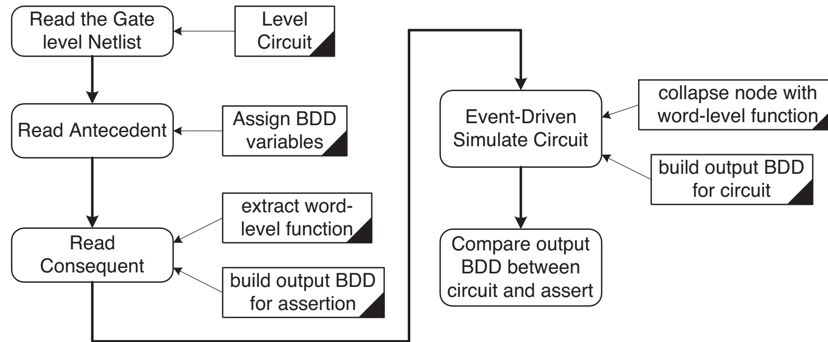

Figure 8. The flow of symbolic simulation with OBDD node collapsing.

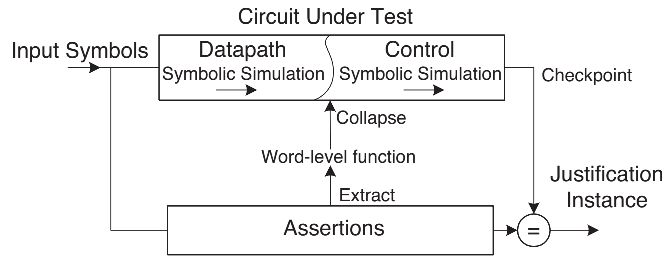

Figure 9. Dual-rail symbolic simulation.

contain only the symbol m_i. On the other hand, with node collapsing, the original equality functions are kept and linked to the new variables m_i. If necessary, m_i can be expanded back to its original equality function. Hence, no information is lost.

6.2. *Dual-Rail Symbolic Simulation*

In our dual-rail simulator, symbolic simulation is applied on both the datapath and the control part. The OBDD node collapsing method is applied on the datapath. This process is shown in Figure 9 (as opposite to that shown in Figure 4).

In our dual-rail symbolic simulation, the results of each signal i are stored as a list of 2-tuples $(D_1^i, V_1^i), \ldots, (D_n^i, V_n^i)$ where each D_j^i is called a *domain* (control part) and each V_j^i is called a *value* (datapath). Both are represented with OBDDs. (D_j^i, V_j^i) can be read as "signal i has the value V_j^i under the domain D_j^i." For all $j \neq k$, (D_j^i, V_j^i) and (D_k^i, V_k^i) are mutually exclusive in terms of the functional spaces they cover.

Initially, each primary input signal i will be assigned with the 2-tuple $(1, V^i)$. In other words, the *domain* at each input is the whole functional space. We note that$(1, V^i) = \{(V^i, 1), (\neg V^i, 0)\}$, and $(D^i, V^i) = \{(D^i V^i, 1), (D^i(\neg V^i), 0)\}$.

During the course of the symbolic simulation, the rules for constructing a two-tuple list are defined over a set of primitives. Figure 10 illustrates some examples. Notice that the *values* in 2-tuples can be moved into the *domains* depending on the primitives encountered. For example, in the case of a MUX, the *value* part $V0$ on the

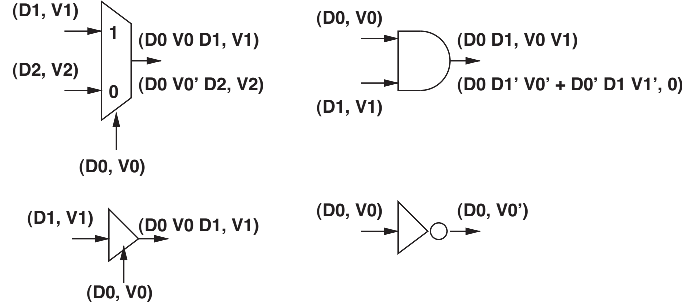

Figure 10. Illustration of 2-tuple list construction rules.

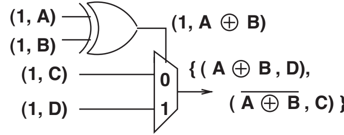

Figure 11. An example of using the 2-tuple list.

select line will be moved to the *domain* part on the output. For a tri-state buffer, a similar operation is performed. We note that for a latch controlled by a latch enable signal, it can be modeled similarly to a MUX where input-0 is given as the original state 2-tuple list, and input-1 is given as the new state 2-tuple list.

Figure 11 shows an example of applying the 2-tuple list. The output of the XOR is used in the *domain* part of the MUX's output 2-tuple list. The input symbols of the MUX (C and D) go into the *value* part of the 2-tuple list.

In the proposed 2-tuple scheme, the definition between control and datapath is not static. For example, given a list $\{(D_1^i, V_1^i), (D_2^i, V_2^i)$, the list can be converted into a single 2-tuple as $(D_1^i + D_2^i, (D_1^i, V_1^i + D_2^i V_2^i)$ because the 2-tuple (D_1^i, V_1^i) and the 2-tuple (D_2^i, V_2^i) are mutually exclusive. Moreover, if $(D_1^i \vee D_2^i = 1)$, then it can further be reduced to $1, (D_1^i V_1^i + D_2^i V_2^i)$. The ability to manipulate a 2-tuple list by exchanging the symbolic values between the *domains* and the *values* implies that the boundary between the control part and the datapath in our dual-rail symbolic simulation is not static (and is not explicitly defined either).

At the end of symbolic simulation, the results from the circuit under check are compared to the results from the model of the assertions for consistency checking. Suppose that results for signal i given by the assertion are $\{(D_1^i, V_1^i), \ldots, (D_n^i, D_n^i)\}$. Suppose that results for the corresponding signal j in the circuit model being checked are $\{(D_1^j, V_1^j), \ldots, (D_m^j, D_m^j)\}$. Then, we first check if $(D_1^i + \cdots + D_n^i) \subseteq (D_1^j + \cdots + D_m^j)$. If this is not true, then the check fails. Otherwise, we further check, for each *domain* $D_k^j, \forall D_l^i, D_l^i \cap D_k^j \neq \phi$, whether or not (D_l^i, V_l^i) is consistent with (D_k^j, V_k^j).

In the first approach, since ATPG needs to enumerate all possible solutions and then call symbolic simulation to run on each sub-case, it may involves repeated

simulation on part of the design. On the contrary, the dual-rail simulation can specify all inputs with symbols and simulate all functionality in one run. With more symbols specified, the complexity in a single run may increase significantly.

7. Additional Experimental Results

In this section, we discuss experimental results on the MMU design using the dual-rail symbolic simulation. We compare two methods: the ordinary symbolic simulator (SS) as described before, and the dual-rail symbolic simulator (SS^2). For each simulator, we discuss the cases with manually optimized initial variable ordering (denoted as "SS-w" and "SS^2-w") and the cases without (denoted as "SS-wout" and "SS^2-wout)."

For these experiments, symbolic variables were assigned to all inputs (and to all initial states of the arrays), and we consider the simulation of the MMU design. All design functionalities were simulated during a single run of the symbolic simulation.

Table 3 shows comparison results on the BAT. For "SS-wout," if no manually optimized initial variable ordering was specified, the symbolic simulation could not handle the case within a reasonable time. We note that for comparing two symbolic vectors, the best ordering is to *interleave* the variables from the two vectors. This idea was used in the manual optimization of the initial variable ordering.

We note that in both simulators, dynamic ordering was implemented. In SS^2, the OBDDs include variables introduced by the node collapsing of equality functions as explained before. It can be observed that the performance of "SS^2" is less dependent on the initial ordering given. On the other hand, the performance of "SS" highly depends on the initial ordering given. Without a good initial ordering, the maximum OBDD size could exceed 20 M.

Table 4 shows results on TLB. Without a good initial ordering, "SS-wout" could not finish the run in 10 min. However, without a good initial ordering, "SS^2-wout" could finish the run very quickly. The last column in Table 4 shows our final results

Table 3. Experimental Results on BAT

	SS-wout	**SS-w**	**SS^2-wout**	**SS^2-w**
Time (s)	Abort	1038.2	6	2
Max OBDD nodes	Too big	2399,140	537	254

Table 4. Experimental Results on TLB and MMU.

	TLB results		**MMU results**	
	SS-w	**SS^2-wout**		**SS^2-wout**
Time (s)	2.2	2	Time (sec)	9.1
Max OBDD nodes	374	170	Total OBDD nodes	16,842

on the MMU. With all inputs being assigned with symbolic variables, the ordinary symbolic simulation could not handle the run even after a significant amount of manual effort were spent to optimize the variable ordering.

8. Conclusion

In this paper, we present two methods for enhancing the capacity of symbolic simulation. We demonstrate the effectiveness of both methods through experiments on the MMU design in Motorola microprocessors. Although both methods utilize similar concepts to avoid OBDD blow-up, we conclude that the *dual-rail* symbolic simulation is an easier and more flexible approach for implementation. In the combined ATPG and symbolic simulation method, symbolic encoding is essential to obtain the efficiency. With OBDD node collapsing, the dual-rail symbolic simulation introduces new variables to replace equality functions. Essentially, the node collapsing method achieves the same efficiency as the CAM encoding method for verifying arrays. The dual-rail symbolic simulation is able to simulate all functionalities in a single symbolic simulation run by assigning symbolic variables to all input signals and to all initial array states.

References

1. Bryant, R. E. Formal Verification of Memory-Circuits by Symbolic-Logic Simulation. *IEEE Transactions on CAD*, vol. 10, no. 1, pp. 94–102, 1991.
2. Ganguly, N., M. S. Abadir and M. Pandey. PowerPC Array Verification Methodology Using Formal Verification Techniques. In *Proceedings of IEEE International Test Conference*, pp. 857–864, Washington, DC, October 1998.
3. Pandey, M., R. Raimi, D. Beatty, and R. E. Bryant. Formal Verification of Powerpc^{TM} Arrays Using Symbolic Trajectory Evaluation. In *Proceedings of 33rd Design Automation Conference*, Las Vegas, NV, 1996.
4. Pandey, M., R. Raimi, R. E. Bryant, and M. S. Abadir. Formal Verification of Content Addressable Memories Using Symbolic Trajectory Evaluation. In *Proceedings of 34th Design Automation Conference*, Las Vegas, NV, 1997.
5. Wang, Li-C., and M. S. Abadir. Experience in Validation of PowerPC Microprocessor Embedded Arrays. *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 15, pp. 191–205, 1999.
6. Narayanan Krishnamurthy, Andrew K. Martin, Magdy S. Abadir, and Jacob A. Abraham. Validating PowerPC Microprocessor Custom Memories. In *IEEE Design and Test of Computers*, pp. 61–76, October–December 2000.
7. Beatty, D. L., R. E. Bryant, and C.-J. H. Seger. Synchronous circuit verification by Symbolic Simulation: An Illustration. In W. J. Dally (Ed.), *Proceedings of the sixth MIT Conference on Advanced Research in VLSI*, pp. 98–112, MIT Press, Cambridge, 1990.
8. Bryant, R. E., D. L. Beatty, and C.-J. H. Seger. Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation. In *28th ACM/IEEE Design Automation Conference*, pp. 397–402, 1991.
9. Seger, C.-J. H. and R. E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in Systems Design*, vol. 6, 147–189, March 1995.
10. Bryant, R. E. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, September 1992.

11. Parthasarathy, G., M. K. Iyer, T. Feng, Li-C. Wang, Kwang-Ting Cheng, and Magdy S. Abadir. Combining ATPG and Symbolic Simulation for Efficient Validation of Array Systems. In *Proceedings of International Test Conference*, Baltimore, October 2002.
12. Feng, Tao, Li-C. Wang, Kwang-Ting Cheng, Manish Pandey, and Magdy S. Abadir. Enhanced Symbolic Simulation for Efficient Verification of Embedded Array Systems. In *Asia and South Pacific Design Automation Conference*, 2003.
13. Wilson, Chris and David L. Dill. Reliable Verification using Symbolic Simulation with Scalar Values. In *37th Design Automation Conference*, June 2000.
14. Wilson, Chris. Symbolic Simulation Using Automatic Abstraction of Internal Node Values. Ph.D. Thesis. Stanford University, October, 2001.
15. Hazelhurst, S. and C.-J. H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDD's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 4, pp. 413–422, April 1995.
16. Aagaard, M. and C.-J. H. Seger. The Formal Verification of a Pipelined Double-Precision *IEEE Floating-Point Multiplier*. In *ACM/IEEE International Conference on Computer-Aided Design*, pp. 7–10, November 1995.
17. Aagaard, Mark D., Robert B. Jones, and Carl-Johan H. Seger. Formal Verification Using Parametric Representations of Boolean Constraints. In *36th ACM/IEEE Design Automation Conference*, 1999.
18. Seger, C.-J. H. Voss—A Formal Hardware Verification System User's Guide. Technical Report 93–45, Department of Computer Science, University of British Columbia, November 1993.
19. *PowerPC^{TM} Microprocessor Family: The Programming Environments*, Motorola Inc., 1994.