Enhanced Symbolic Simulation for Efficient Verification of Embedded Array Systems

Tao Feng*, Li-C. Wang*, Kwang-Ting Cheng*, Manish Pandey†, Magdy S. Abadir††
* Department of ECE, UC-Santa Barbara † Verplex Systems, Inc.
†† ASP High Performance Design, Motorola, Inc.

Abstract— In the past, Symbolic Trajectory Evaluation (STE) was 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 using a "dual-rail" symbolic simulation scheme to facilitate the application of STE proof methodology for verifying array systems. The proposed scheme implicitly partitions a given design into control domain and datapath domain, and symbolic simulation is carried out on both domains. With this scheme, the run-time OBDD sizes during the symbolic simulation for each domain 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 an ordinary symbolic simulator was used in the STE proof process.

I. 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 for performance optimization. A typical array system contains multiple array blocks, and interactions among these blocks can be complex and hard to verify.

Formal techniques such as STE assertion-based symbolic simulation have been shown to be effective for verifying arrays in terms of both functional verification and structural equivalence. In the STE proof methodology, symbolic simulation is used as the underlying engine [1] which utilizes *Ordered Boolean Decision Diagrams* (OBDDs) [2] to represent ternary logic functions.

Past SET-based techniques could usually be applied for verifying individual array blocks [3, 4, 5]. With special *symbolic encoding* techniques, it can also be efficient for verifying arrays which are content addressable in nature [6]. Nevertheless, when verifying an operation that involves interactions among multiple array blocks, the OBDD sizes could often blow up. To overcome the problem, our earlier work incorporates an ATPG-style decision procedure into the symbolic simulation framework [7]. In that approach, the symbolic simulation process is partitioned into separate and simpler sub-processes by ATPG assigning constant values to a set of selected control signal lines. And, during the symbolic simulation of each sub-process the OBDD sizes can be minimized with *symbolic encoding*.

With the same objective of applying the STE method at the system level, in this paper, we propose a different approach to avoid the OBDD size blow-up problem. Our current approach is different from the previous approach [7] in terms of three aspects: (1) Instead of explicit partitioning a design into control and datapath, our current approach adopts a partition scheme implicitly in the symbolic simulation process. (2) Instead of using a decision procedure such as ATPG to handle the control part [7], we use symbolic simulation on both the control and the datapath. (3) Instead of utilizing symbolic encoding to minimize OBDD sizes on the datapath, we adopt a simpler *node collapsing* method that gives the same efficiency.

We call our current approach the *dual-rail* symbolic simulation scheme because essentially symbolic simulation is carried out on control and datapath separately. With this new method, the OBDD sizes in the symbolic simulation can be limited. Our goal is to demonstrate that this new strategy can also verify multiple array blocks together as a single system, as opposed to verify individual blocks separately in a system. Moreover, we argue that our method is more general and more efficient than the approach proposed before in [7].

II. Comparison to Prior Work

STE has been used to verify memory arrays such as on-chip caches and register files. In [3], authors used the *Voss* STE [8] system to verify a multi-ported register file unit and a data tag unit in Motorola *PowerPC* ™ microprocessors. In [6], 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 verification was performed on the actual designs. In [4], various validation methods, including STE were compared based upon design error injection and simulation. It was shown that although STE approach was able to verify array blocks, the quality of verification could be highly dependent on the *assertions* supplied to the STE process. Therefore, to ensure high quality, automatic generation of assertions [9] was proposed. In a recent work [5], 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 not applied at the system level to verify multiple array blocks as a whole. For 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.

In an earlier work [7], we proposed to combine an ATPG-style decision procedure with the OBDD-based symbolic simulator for array system validation. The central ideas included: 1) partitioning a given design into control and datapath domains, and 2) applying the ATPG

0-7803-7659-5/03/$17.00 ©2003 IEEE.

procedure to assign constant values on the controls, and then symbolic simulation to handle the datapath based upon the constant-assigned control values. In essence, to verify that a design satisfies a given assertion, the assertion is divided by ATPG-assigned constant values into a sequence of simpler sub-assertions. These sub-assertions are then to be checked by symbolic simulation independently. With this combined strategy, blow-up in OBDD sizes can be avoided and STE-based assertion verification can be applied to a larger system [7].

The idea of using multiple engines (such as combining SAT solver and OBDD-based symbolic simulation) was not new. For example, Wilson *et. al.*, [10, 11] proposed using a combined SAT and OBDD-based symbolic simulation approach to eliminate run-time OBDD memory blow-up. Authors in [12] proposed using arithmetic and Boolean ATPG solver to verify arithmetic circuits. Authors in [13] proposed a hybrid approach that 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. Authors in [14] used the hybrid approach to verify a radix-eight, pipelined, IEEE double-precision floating point multiplier.

In this paper, we propose a more general solution than that in [7]. Instead of using a decision procedure and the symbolic simulation, our approach utilizes symbolic simulation for both control and datapath. Instead of static partitioning a design into control and datapath, the partitioning is carried out in the symbolic simulation process. This allows a more flexible implementation to dynamically adjust the partition boundary. Moreover, because the assertion is checked by the *dual-rail* symbolic simulation in one pass, as opposed to running symbolic simulation multiple times for multiple sub-assertions [7], the new approach can avoid duplicate symbolic simulation work on the same portion of the design. Potentially, this can enhance the overall verification efficiency [9]. The effectiveness of our techniques will be demonstrated through experiments on an MMU design from Motorola high-performance microprocessors [15].

III. Background

Symbolic Trajectory Evaluation (STE) [1] is a formal verification technique that is based on ternary symbolic simulation . 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 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

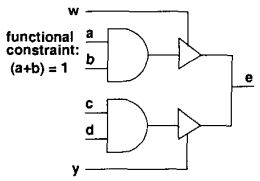

Figure 1: An Illustrative Example

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 "$\cdot$" to specify logic AND in the 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.

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

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

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

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 [5]. 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.

In our work, we assume that the symbolic simulator is cycle-based. For the simplicity of discussion, first we assume that 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. Later, with our newly proposed method, we will see that the "assertions" do not need to be specified in the form described above. Instead, the golden model can be any simulatable model in the form of another RTL or an assertion model. Hence, our work can also reduce the burden of manual creation of assertions and hence, improve the overall verification quality.

IV. CAM: An Illustration Example

In this section, we use a simple content addressable memory (CAM) to illustrate why an ordinary symbolic simulation can be inefficient. Figure 2 depicts such a CAM example.

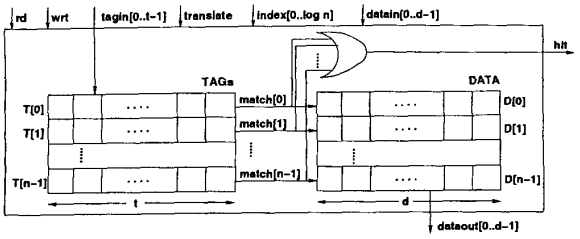

Figure 2: A CAM: Tag size $= t$, No. of entries $= n$, and Data size $= d$

Three operations are defined on this CAM: *read*, *write*, and *translate*, controlled by the three inputs *rd*, *wrt*, and *translate*, respectively. When performing a *read* or a *write*, the CAM behaves like a regular array where both rows in the TAGs and DATA arrays are accessed or updated based upon the input address *index*[], input tag *tagin*[], and input data *datain*[]. For *translate* operation, the CAM performs an associative read. 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, $match[j] = 0$. The *hit* signal indicates that there is a match in the tag.

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

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

where *nomatch* indicates the condition that $\overrightarrow{T} \neq \overrightarrow{T_i}$ for all $i = 0,1,2,3$. And *matchonly*0 indicates the condition that $\overrightarrow{T}$ matches only $\overrightarrow{T_0}$ (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* [6]. 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 ($= t \times n + t$).

IV-A. CAM Encoding

To reduce the number of symbolic variables, symbolic encoding can be applied to encode the fact that for each way, the comparison can only have two outcomes: match and mismatch. The fundamental idea of using symbolic encoding to reduce the number of symbolic variables was first proposed in [6]. Assume $tag = \overrightarrow{T} = [t_0, \ldots, t_{w-1}]$ where w is the tag width. In order for the way 0 tag $\overrightarrow{T_0}$ to mismatch with *tag*, $\overrightarrow{T_0}$ should be consistent with one of the following tenary vectors: $(\neg t_0, X, \ldots, X)$, $(X, \neg t_1, X, \ldots, X)$, ..., $(X, \ldots, X, \neg t_{w-1})$. On the other hand, in order for $\overrightarrow{T_0}$ to match with *tag*, $\overrightarrow{T_0}$ should be the same as $\overrightarrow{T}$. In other words, assuming that the input *tag* is given with the symbolic vector $\overrightarrow{T}$, we can encode the content of way 0 tag $T[0]$ as the following.

$$\begin{gathered}T[0][0] := ((when(m_0' \wedge (I_0 = 0)))(\neg t_0)) \wedge ((when(m_0)(t_0)) \\ T[0][1] := ((when(m_0' \wedge (I_0 = 1)))(\neg t_1)) \wedge ((when(m_0)(t_1)) \\ \ldots \\ T[0][w-1] := ((when(m_0' \wedge (I_0 = \\ w-1)))(\neg t_{w-1})) \wedge ((when(m_0)(t_{w-1}))\end{gathered}$$

In addition to the symbols $t_0, \ldots, t_{w-1}$ used in $\overrightarrow{T}$, notice that we introduce a new symbolic vector I_0 and a new symbolic variable m_0 to encode the all possible initial states in $T[0]$. I_0 is $\lceil \log w \rceil$ wide. Its purpose is to indicate which bit has the mismatch. m_0 is to indicate if way 0 tag $T[0]$ matches the incoming *tag* or not. Then, it can be easily verified that $(when(I_0 < w))$, the comparison between *tag* and $T[0]$ will always result in two outcomes: match (m_0) and mismatch (m_0'). Similar encoding can be applied to $T[1], T[2]$, and $T[3]$ (using additional I_1, I_2, I_3 and m_1, m_2, m_3).

Why the above encoding scheme is more efficient for symbolic simulation? One reason is that here we use less number of variables to represent all possible states in $T[0]$ (instead of w variables, here we use $\lceil \log w \rceil + 1$ variables). Most importantly, observe that after symbolic simulation, the OBDD on the $match[0]$ signal contains only one variable m_0 and hence, is of the minimal size. Essentially, the above encoding scheme implicitly "re-start" the symbolic simulation at the signal line $match[0]$ by introducing a new variable m_0. As a result, the verification on the circuit before and after the $match[0]$ signal is separated. Hence, the complexity can be dramatically reduced.

IV-B. Split Assertion by Decision Procedure

When verifying multiple array blocks together, a CAM is usually embedded in a bigger design and an assertion can be much more complex. The symbolic encoding may not be easily applicable. This is because in order to encode the content, we need to know, in advance, the symbolic *tag* inputs given to the CAM. Moreover, we may need to know what would be the expected outcome of the CAM in order to avoid the potential false negative problem (explained later).

Another way to solve the OBDD size blow-up problem is to incorporate a decision procedure such as ATPG into the symbolic simulation [7]. For example, we can partition the CAM design in such a way that the OR gate to generate the *hit* is handled by an ATPG decision procedure instead of symbolic simulation. For instance, to check a CAM hit assertion, conceptually ATPG would set a 1 at the *hit* of the *assertion-synthesized* circuit (the same assertion viewed as a circuit), and inverse the objective to set $hit = 0$ on the circuit under check. Then, ATPG will enumerate all decisions on the assertion circuit to justify the goal by assigning $100\cdots0$, $010\cdots0$, $110\cdots0$, $0010\cdots0$, ..., $111\cdots1$ to the OR inputs $match[0]\cdots match[n-1]$. In each case, since the match outputs are all specified as constants (i.e. the expected outcome is known), we can use CAM encoding techniques to encode the content of the CAM [7]. Then, symbolic simulation is called to justify that in each case, $hit = 0$ is indeed impossible in the circuit under check due to the constraints imposed by the assertion circuit. In other words, we have proved that $hit = 1$ on the assertion implies $hit = 1$ on the circuit under check. Conceptually, the decision procedure partitions a given assertion into a sequence of much simpler sub-assertions to be more efficiently handled by the symbolic simulation. We note that in order for an ATPG to work smoothly with the symbolic simulation in the way described above, additional techniques are needed [7].

IV-C. Problems and Motivations

Consider the verification of the MMU design in Motorola high-performance microprocessors [15] as illustrated in Figure 3. Motorola microprocessor MMU contains a 64-entry, two-way set-associative, Translation Look-aside Buffers (TLB) that provide support for demand-paged virtual memory address translation and variable-sized block translation. MMU supports block address translation through the use of block address translation (BAT) array containing four entries. Up to 15 bits of the effective addresses are compared simultaneously with all four entries in the BAT during the translation. If the

address is present in the BAT, the *bathit* 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.

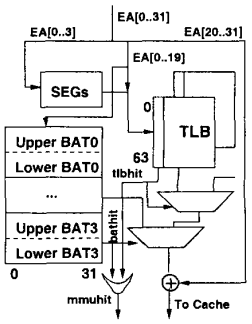

Figure 3: Block Diagram of MMU [15]

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, which is similar to the CAM design example described in Figure 2.

Verification of the BAT can be done efficiently with the CAM encoding method described before. However, individual assertion is required, i.e. verification of *read*, *write*, and *translate* operations must be separated. In other words, given a BAT assertion, depending on what to be checked in the consequent, certain CAM encoding is used to simplify the symbolic simulation. This prevents us from applying the CAM encoding idea to verify all three operations together in one run of the symbolic simulation.

Why verifying all operations together can be useful? Consider verification of a BAT RTL model against another RTL model that has been slightly modified. What we can do is to supply symbolic inputs to the original BAT model, symbolically simulate the model, and then collect results at its outputs. The collected results (OBDDs) essentially can serve as the consequent part to be checked on the modified BAT model (where the same symbolic inputs are given to this model). For this application, what we really need is an efficient way to verify all BAT operations in a single run of the symbolic simulation (by setting up all control input signals with symbolic inputs as opposed to constant values). Since the CAM encoding schemes only encode for *translate* behavior, not *read* and *write*, they cannot be applied.

Now consider the verification of an MMU assertion involving BAT, SEG, and TLB together. The proposed CAM encoding schemes cannot be easily applied either. When verifying BAT alone, results are checked at *bathit* output. In that case, the "re-start" of the symbolic simulation given by the CAM encoding at $match[i]$ signals is not a problem because no re-converging fanout is involved. In the whole MMU, *ea* is used in both BAT and TLB, and the results converge at the final *mmuhit* signal. Hence, when using CAM encoding schemes in both BAT and TLB, individually the verification is correct, but collectively it is not correct because we cannot verify the fact that both associative reads share some part of the same effective address *ea*. To understand this, consider the results at *bathit*. With CAM encoding, the OBDD at *bathit* consists of four variables m_0, m_1, m_2, m_3. This OBDD will be ORed with the result at *tlbhit* to produce the final result at *mmuhit*. Therefore, the OBDD at *mmuhit* contains no variable from *ea*. Consequently, there is no way to check if *bathit* and *tlbhit* are generated based upon the same *ea* or different *ea*s.

In general, if a datapath contains re-converging fanouts and encoding methods are used to simplify the symbolic simulation on one fanout path, then the verification may be false-negative.

CAM encoding, however, can be used effectively in a combined ATPG and symbolic simulation strategy [7]. This is because the ATPG decision procedure can break the re-converging fanouts by assigning constant values to ensure only one fanout path is activated at a time. The drawback in the combined strategy is the need to enumerate the selected signals to exhaust all space. As explained earlier, to justify $bathit = 1$, symbolic simulation has to repeat $2^4 - 1 = 15$ times where in each time some portion of the design is simulated repeatedly.

In summary, what we want is an approach that can take advantages of the ideas employed in both CAM encoding and the combined ATPG and symbolic simulation strategy and yet, can avoid their drawbacks. The proposed approach should also be able to efficiently handle all operations of a design in a single symbolic simulation run.

V. Dual-Rail Symbolic Simulation

The fundamental idea in [7] was to separate control and datapath so that the control part could be handled by an ATPG decision procedure and datapath can be handled by the symbolic simulation. The fundamental idea here is to employ a separate symbolic simulation process for the control part as well. Since symbolic simulation is used on both control and datapath, the boundary of control and datapath does not need to be fixed and can be dynamically adjusted.

In our symbolic simulation, the results of each signal i are stored as a list of 2-tuples $(D^i_1, V^i_1), \ldots, (D^i_n, V^i_n)$ where each D^i_j is called a *domain* and each V^i_j is called a *value*. (D^i_j, V^i_j) is read as "signal i has the value V^i_j under the domain D^i_j." For all $j \neq k$, (D^i_j, V^i_j) and (D^i_k, V^i_k) are mutually exclusive in terms of the functional spaces they cover. We note that D^i_j and D^i_k may not be mutually exclusive though. This differentiates our scheme from the 2-tuple scheme in [9].

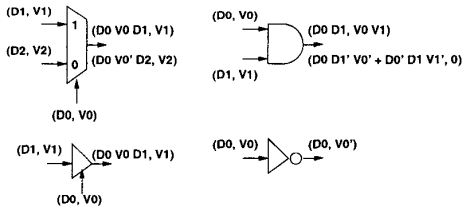

Figure 4: Illustration of Domain and List Merging Rules

Initially, each 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 our symbolic simulation, *values* can be moved into *domains* depending on the primitives encountered. For example, Figure 4 illustrates some situations where this could happen. Notice that in the case of a MUX, the *value* part $V0$ on the select line will be moved to the *domain* part on the output. For 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.

In general, during the dual-rail symbolic simulation, the computed function for a signal line is represented by a list of 2-tuples. Hence, when symbolically simulating a logic operation of two signals, *merging rules* are required to process the 2-tuple lists. In the figure, rules for AND and inverter are illustrated.

Conceptually, the initial partition between the control (*domain*) and the datapath (*value*) is formed by control signals to the primitives such as MUX, tri-state, latches, etc. However, sometimes a control signal may not appear as the control line to such a primitive. Figure 5 illustrates the problem.

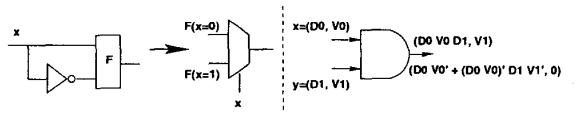

Figure 5: Input Signal Treated as a MUX select

In a real design, it is unrealistic to expect that all controls are connected to the control lines of some well-defined primitives. Hence, it is necessary to have a pre-processing step to recognize some important control signals. Theoretically, any signal (x) whose positive (x) and negative ($\neg x$) forms are used in the computation of a single-output function F can be treated as a control signal for F. Figure 5 illustrates the situation. In this case, $x = (D0, V0)$ is first converted into $x = \{(D0V0, 1)(D0(\neg V0), 0)\}$ before any logic operation is performed with other signals. This is illustrated in the figure by an AND with another signal y. Notice that at the output of the AND gate, the *value* $V0$ appears only in the domain part of the 2-tuples.

In the proposed 2-tuple scheme, the definition between control and datapath is not static. For example, suppose $\{(D_1^i, V_1^i), (D_2^i, V_2^i)\}$ are obtained for a signal 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 $(1, D_1^i V_1^i + D_2^i V_2^i)$. In other words, we can move *domains* into *values* in the 2-tuple representation. The ability to manipulate a 2-tuple list by moving back and forth the *domains* and the *values* in the 2-tuples implies that the boundary between control and datapath in our dual-rail symbolic simulation scheme is not static. This differentiates our approach from the static partitioning method used in the combined ATPG and symbolic simulation strategy proposed in [7]. Our approach allows more flexibility to dynamically adjust this boundary during run time.

V-A. Node Collapsing

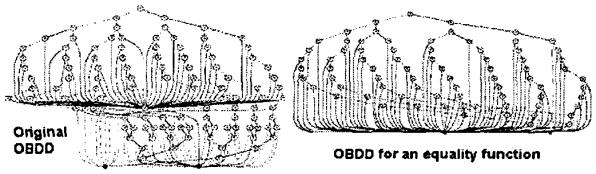

Figure 6: Original OBDD and The Equality Function

Recall that the CAM encoding schemes are efficient because they abstract the behavior of a CAM by introducing new variables (m_i) at each $match[i]$. In our approach, the simulator introduces a new variable when an equality function ($A = B$) of two symbolic input vectors (A, B) is identified during the course of symbolic simulation.

For example, suppose at the beginning of symbolic simulation, three symbolic vectors of equal lengths are defined, denoted as T_0, T_1, T_2. Then, during the course of the simulation, the simulator will check if the computed results of a signal *contain* one of the equality functions $T_0 = T_1$, $T_0 = T_2$, and $T_1 = T_2$ as a sub-function. Take $T_0 = T_1$ as an example. The equality function $T_0 = T_1$ is essentially the output result from a word-level comparator primitive $COMP(T_0, T_1)$. Since the comparator primitive may not be well defined in a real design and also T_0 and T_1 may have been combined with other control symbols before entering the comparator, the simulator needs to constantly monitor the computed results to identify that an equality function has been computed.

In our implementation, we use various heuristics to minimize the need for constantly monitoring computed results. However, when it does happen, Figure 6 and 7 illustrate the operations that the simulator performs. In Figure 6, the left-hand side shows an OBDD corresponding to the simulated result for an internal signal. The right-hand side is the OBDD corresponding to the equality function such as $T_0 = T_1$ to be compared with. In Figure 7, the left-hand side shows the signal OBDD by re-ordering the variables in such a way that variables in T_0 and T_1 appear at the bottom of the OBDD. Then, since the simulator utilizes a *Reduced OBDD* strategy where each unique function is stored only once, a simple pointer check can determine if the OBDD of "$T_0 = T_1$" is contained as a sub-graph in the left-hand side OBDD or not. In this case, this is true and hence, the left-hand side OBDD can be simplified by introducing a new variable (the shaded node) to replace the entire "$T_0 = T_1$" sub-graph. The resulting OBDD is shown on the right-hand side of the figure.

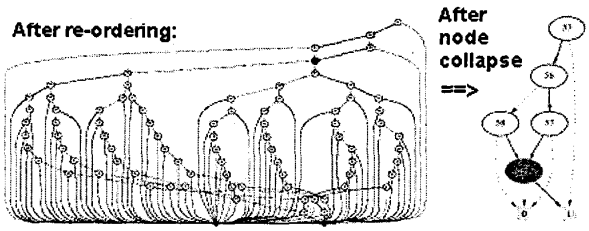

Figure 7: Identify Containment of the Equality Function by Re-Ordering

In our symbolic simulation, new variables are introduced to replace the OBDD sub-graphs that represent equality functions of pairs of symbolic vectors. For CAM, this achieves the same abstraction effect as the CAM encoding schemes do. However, since the original equality functions are kept and linked to the new variables, no information is lost. Hence, if necessary such a variable can be expanded back to its corresponding equality function later on. This avoids the false-negative problem described earlier.

V-B. Consistency Checking

At the end of symbolic simulation, results from the model under check are compared to the results from the golden model (can be an RTL or an assertion). Suppose that results for signal i in the golden model are $\{(D_1^i, V_1^i), \ldots (D_n^i, V_n^i)\}$. Suppose that results for the corresponding signal j in the model being checked are $\{(D_1^j, V_1^j), \ldots, (D_m^j, V_m^j)\}$. 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^i, $\forall D_l^j, D_l^j \cap D_k^i \neq \phi$, whether or not (D_l^j, V_l^j) is consistent with (D_k^i, V_k^i).

VI. Verification of the MMU

In this section, we discuss experimental results on the Motorola MMU design. In each experiment, we compare two methods: the ordinary symbolic simulator (SS) and our dual-rail symbolic simulator (SS^2). For the ordinary simulator, we use a Motorola in-house tool that was modified from *Voss* [8]. For each simulator, we discuss the case with manually optimized initial variable ordering and the case without. In the former case, we denote the two methods as "SS-w" and "SS^2-w." In the later case, we denote the two methods as "SS-wout" and "SS^2-wout." In addition, we also consider the case of using CAM encoding schemes and denote it as "SS-encode."

All our experiments were run on a Pentium 4 1.5G machine running Linux Mandrake 2.4.8-26mdk with 512M memory. For all experiments, symbolic variables were assigned to all inputs and no constant value was used (unless otherwise specified). Hence, all operations were intended to be verified during a single run of the symbolic simulation.

	SS-wout	SS-w	SS-encode*	SS^2-wout	SS^2-w
time (sec)	abort	1038.2	0.3	6	2
max OBDD nodes	too big	2399140	<500	537	254

*only consider *translate* operation

TABLE I: EXPERIMENTAL RESULTS ON BAT

Table I 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.

The maximum number of OBDD nodes computed for a signal during symbolic simulation (max OBDD nodes) is also shown as a complexity measurement. We note that in both simulators, dynamic ordering was implemented. In SS^2, the OBDDs contain variables introduced by node collapsing of equality functions as explained before.

In the case of "SS-encode," since the CAM encoding method could not handle all operations simultaneously, only *translate* operation (associative read in CAM) was simulated. As expected, for this operation, the "SS-encode" performed very well.

It can be observed that the performance of "SS^2" is less dependent on the initial ordering given, and is not too far from the method "SS-encode." On the other hand, the performance of "SS" highly depends on the initial ordering given. With a good initial ordering it could finish the run. Without a good initial ordering, it could not finish the run. In the later case, the OBDD size exceeded 20M.

	SS-wout	SS-w	SS^2-wout	SS^2-w
time (sec)	abort	2.2	2	0.01
max OBDD nodes	too big	374	170	103

TABLE II: EXPERIMENTAL RESULTS ON TLB

Table II shows results on TLB which is a 2-way array, each with 64 entries. Tags are 35-bit wide in this 2-way associative organization. The effect of a good initial ordering on the ordinary symbolic simulation "SS" can clearly be observed. Without a good initial ordering, "SS-wout" could not finish the run. With a good initial ordering, "SS-wout" could finish the run very quickly.

	Time (sec)	Max OBDD nodes
SS^2-wout (all functionality)	9.1	842
SS/ATPG(1 assertion only)**	8.3	not reported

*SS aborts in this case

**The assertion considers only for $bathit = 0, tlbhit = 1$

TABLE III: EXPERIMENTAL RESULTS ON MMU

Table III shows our final results on the MMU. With all inputs being given symbolic variables, the ordinary symbolic simulation could not handle the run even after a significant amount of manual effort were spent to optimize variable ordering. In the table, we also copy the "SS/ATPG" (the combined symbolic simulation and ATPG strategy) result from [7] where the MMU assertion considered only the case by assigning $bathit = 0, tlbhit = 1$. As explained before, to justify $bathit = 1$ the ATPG decision procedure would enumerate all 15 cases on the $match[0], match[1], match[2], match[3]$ signals shown in the CAM example (Figure 2). Hence, it would take a much longer time for "SS/ATPG" to complete the same work as that was completed by the "SS^2-wout" in just 9.1 seconds.

VII. CONCLUSION

In this paper, we present a novel *dual-rail* symbolic simulation approach for efficient system-level verification of embedded memories. Our approach utilizes a 2-tuple simulation scheme to enable partitioning between the control and the datapath. Unlike the combined ATPG and symbolic simulation approach proposed earlier [7], the partitioning in our scheme does not have to be static. This gives more flexibility in our implementation for optimizing performance. During the course of simulation, our approach introduces new variables to represent equality functions on pairs of symbolic input vectors. Essentially, our simulator can achieve the same efficiency as the CAM encoding for verifying individual CAMs, and does not have the false-negative problem for verifying other array designs in general. The effectiveness of our proposed techniques is demonstrated through experiments on the MMU design in Motorola high-performance microprocessors. For the MMU, our approach is able to simulate all operations in a single symbolic simulation run with symbolic variables supplied to all input signals. This feature helps to minimize the manual effort of creating individual assertion for each operation during the verification.

REFERENCES

[1] 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.

[2] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, Sep 92.

[3] M. Pandey, R. Raimi, D. Beatty, and R.E. Bryant. Formal verification of powerpc^{tm} arrays using symbolic trajectory evaluation. In *33rd DAC*, Las Vegas, NV, 1996.

[4] 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.

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

[6] M. Pandey, R. Raimi, R.E. Bryant, and M.S. Abadir. Formal verification of content addressable memories using symbolic trajectory evaluation. In *34th DAC*, Las Vegas, NV, 1997.

[7] G. Parthasarathy, 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 *Proc. International Test Conference*, Baltimore, Oct 2002

[8] 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.

[9] 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.

[10] 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*, Nov 2000

[11] Chris Wilson and David L. Dill. Reliable Verification using Symbolic Simulation with Scalar Values, *37th DAC*, June 2000

[12] 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.

[13] 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, 1995.

[14] 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.

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