Formal verification of PowerPC^{TM} arrays using symbolic trajectory evaluation *

Manish Pandey[1] Richard Raimi[2] Derek L. Beatty[2] Randal E. Bryant[1]

[1]School of Computer Science, Carnegie Mellon University Pittsburgh, PA 15213
[2]Motorola Inc., 6501 William Cannon Drive West, Austin, TX 78735.

Abstract

Verifying memory arrays such as on-chip caches and register files is a difficult part of designing a microprocessor. Current tools cannot verify the equivalence of the arrays to their behavioral or RTL models, nor their correct functioning at the transistor level. It is infeasible to run the number of simulation cycles required, and most formal verification tools break down due to the enormous number of state-holding elements in the arrays.

The formal method of symbolic trajectory evaluation (STE) appears to offer a solution, however. STE verifies that a circuit satisfies a formula in a carefully restricted temporal logic. For arrays, it requires only a number of variables approximately logarithmic in the number of memory locations. The circuit is modeled at the switch level, so the verification is done on the actual design.

We have used STE to verify two arrays from PowerPC microprocessors: a register file, and a data cache tag unit. The tag unit contains over 12,000 latches. We believe it is the largest circuit to have been formally verified, without abstracting away significant detail, in the industry. We also describe an automated technique for identifying state-holding elements in the arrays, a technique which should greatly assist the widespread application of STE.

1. Introduction

In this paper we report on using Symbolic Trajectory Evaluation (STE) to verify on-chip memory arrays from PowerPC microprocessors. Arrays include circuits such as multi-ported register-files, instruction and data caches and cache tag units. These circuits typically consist of a Static Random Access Memory (SRAM) core embedded within complex logic. Such units are generally designed at the transistor-level and have non-trivial internal timing, including self-timed components.

Verification of on-chip arrays has been a weakness in the verification strategies of many companies. Behavioral or RTL models of arrays are usually simulated as part of the full-chip verification effort. These simulation results then need to be related to the actual array implementations. In recent years, formal verification tools for comparing RTL models to gate level netlists have come into widespread use[7]. Most use Ordered Binary Decision Diagram (OBDD) representations of Boolean functions [4], and utilize the canonicity of OBDDs to determine equivalence. It is now common in the industry to simulate at the RTL level, and depend on such Boolean comparison tools to guarantee equivalence of lower-level implementations.

Arrays have not fit well into this strategy, however. Translating transistors into combinational logic and latches can fail to capture the complex timing in arrays. The state explosion problem poses a greater difficulty, however. Arrays contain enormous numbers of storage elements, and a Boolean comparison tool will naively attempt to build OBDDs of functions which depend on all the storage bits. This is usually not feasible.

Recently, at the joint IBM-Motorola PowerPC microprocessor design center, Somerset, the formal verification technique of Symbolic Trajectory Evaluation (STE) has been applied to arrays. STE offers the following advantages:

1. Array properties can be verified using a number of variables approximately logarithmic in the number of array nodes, ameliorating the state explosion problem.
2. STE tools utilize switch-level simulation, allowing accurate modeling of actual transistor behavior.
3. Arrays usually have concise, well-understood specifications. STE tools can directly verify adherence of the transistor or RTL model to these specifications.

We used the Voss STE system [11] in our work. Voss provides a powerful, functional language interface to the STE verifier, called FL. FL is a strongly-typed polymorphic functional language, similar to ML [9]. Voss represents Boolean functions with OBDDs, making Boolean function manipulation particularly fast and convenient.

2. Preliminaries

STE [10] is a descendant of symbolic simulation [5]. A symbolic simulator propagates symbolic variables through a circuit network, in addition to logic constants. For symbolic simulation to be efficient, a compact format for Boolean functions is needed. The development of Ordered Binary Decision Diagrams (OBDDs) in the late 1980's provided such a format [4]. With that advance came the desire to integrate symbolic simulation with a rigorous, formal proof procedure. Bryant and Seger developed the theory of Symbolic Trajectory Evaluation towards that end [10].

In STE, properties of circuits are expressed in a restricted temporal logic. Formulae in the underlying logic can be simple predicates (e.g., 'node x is 1') or conjunctions of these. Such formulae can be operated on by a *next time* operator (e.g., "node x is 1 on the next time step"), or qualified by *domain restriction* (e.g., "node x is 1 when function E is true"). These latter are also formulae. An assertion is an implication between two formulae.

* This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330 and in part by a grant from Motorola Inc.

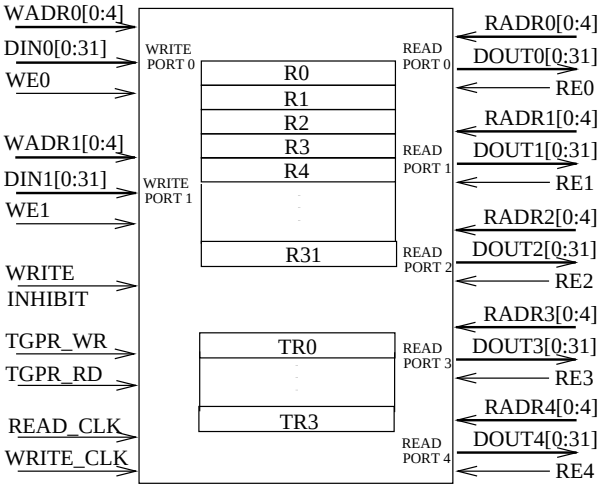

Figure 1: Multi-ported Register File Unit.

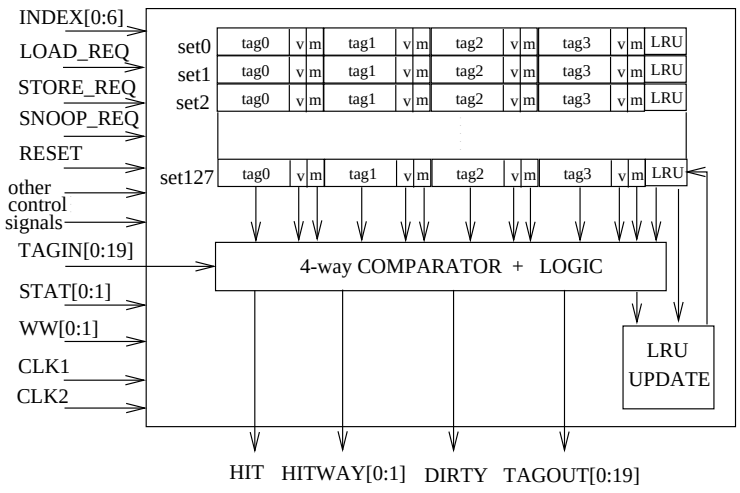

Figure 2: Data Cache Tags Unit

STE provides an algorithm for proving such assertions valid. It is a model-checking algorithm, in that it checks whether a system is a model of an assertion in the logic. However, it should not be confused with CTL model-checking [6]. STE lacks the expressiveness of CTL, e.g., eventuality properties are not expressible, nor is existential path quantification available. Use of STE is limited to applications wherein the properties to be verified are expressible in its (rather limited) logic, and, in addition, the set of starting states for the verification can be safely assumed to be reachable states. Unlike CTL model-checkers, STE tools do not calculate the reachable state set of a circuit. For arrays, these restrictions are acceptable. It is generally sound to assume that array nodes can hold arbitrary bit combinations. And, even in cases where this is not so, the state invariants can usually be manually derived and the array properties verified under these invariant conditions. Additionally, the behavior of memories can generally be expressed in the STE logic.

To use STE, properties of circuits are expressed as assertions of the form $A \implies C$, where *A* is termed the *antecedent*, and *C* the *consequent*. Intuitively, the antecedent defines initial settings and input stimuli, while the consequent defines expected results. The symbolic simulation engine simulates the symbolic patterns of the antecedent, and simulation results are compared to the consequent.

The intuitive sense of STE is that it proves that the behavior exhibited by symbolic simulation of the antecedent is one of the possibly many behaviors consistent with the consequent, proving this for any assignment of values to the variables involved. The reader is referred to [10] for more detail.

3. PowerPC array circuits

The first circuit we verified was a relatively simple multi-ported register file unit of a PowerPC microprocessor. The second circuit is the tags unit for a data cache circuit from a recent PowerPC design.

3.1. Multi-ported register file

Figure 1 shows a high-level view of the register file, which has 2 identical write ports and 5 identical read ports. When READ_CLK is high, the register file does a read operation and when WRITE_CLK is high, the register file does a write operation. The READ_CLK and the WRITE_CLK signals are mutually exclusive. The environment guarantees that the two write addresses are always different.

The register file contains 36 registers of 32 bits each, arranged in two banks, R0-R31 and TR0-TR3. During a write operation, when TGPR_WR is low, the writes go to one of R0-R31 as specified by the 5-bit address for each write port. When TGPR_WR is high, the writes go to one of TR0-TR3 based on the two least significant address bits. The environment is supposed to keep the middle address bit (bit 2) at 0, when the TGPRs are to be written. WRITE_INHIBIT when high prevents any writes from occurring. Also, each port has a write enable signal (WE0, WE1).

The read ports also have read enable signals (RE0,..., RE4). When TGPR_RD is low, the five address bits select a register from the first bank. When TGPR_RD is high, the lowest two address bits select a register from the second bank, and bit 2 of the address must be low for the read to be successful. If a read does not occur on a port, or if bit 2 of the address is high when TGPR_RD is high, then the port's data output stays precharged high.

3.2. Data cache tags unit

The data cache tags (DTAG) circuit, shown in figure 2, contains 128 4-way-associative sets. Each set contains 4 tags of 20 bits each, and each tag has one valid and one modified (dirty) bit. Also, each set contains 6 least-recently-used (LRU) bits which record the access history of its four ways.

In a typical operation, a 7-bit index at the INDEX input selects one of the 128 sets, and the 20-bit tag at TAGIN is compared in parallel with all four tags in the selected set. If a tag matches, then the HIT signal goes high and the LRU bits are updated to reflect that the matched way is most recently used. HITWAY indicates which of the four ways is hit. If none of the four tags match, the HIT signal remains low, and the least recent tag appears at TAGOUT (for cache replacement).

Other important operations are the reset and the tag write operations. In the reset operation, the RESET signal resets the DTAG unit by zeroing all valid, modified and LRU bits. In the tagwrite operation, the tag value at TAGIN and the valid and modified bit values at STAT are written into a way selected by WAYSEL of a set specified by SLOW_INDEX.

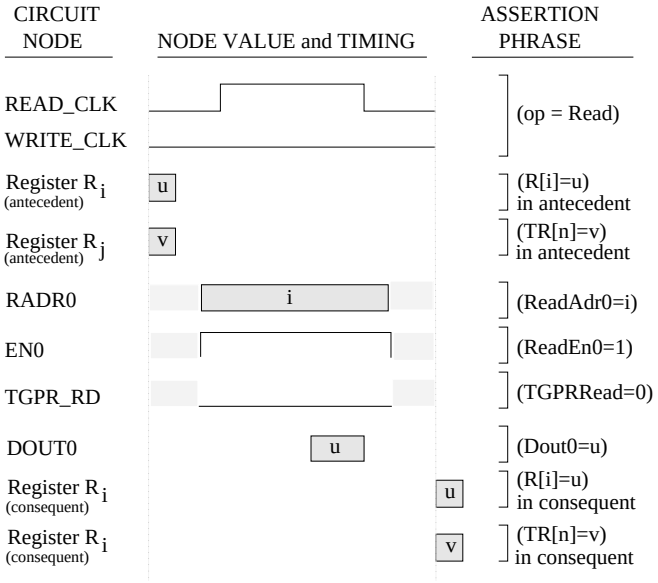

Figure 3: Implementation mapping for the register-file.

4. Verification methodology

In verifying the arrays, we structured our specifications into 2 parts: a set of high level assertions over an abstracted system state, and an implementation mapping that relates that abstracted state to circuit state. The set of assertions is defined by the set of operations that the array can perform. Each assertion gives the conditions required for doing one operation, and the conditions guaranteed as a result of it. The implementation mapping converts the abstracted state into constraints on signals in the circuit over time. Structuring specifications in this way keeps the most critical part of the specification—the abstract description of the desired behavior—simple, clear, and free of implementation-specific details. This methodology [1, 2] is not directly supported by Voss. However, we disciplined our use of the FL language to write our specification in a hierarchical manner and separated the abstract assertions from the implementation mapping.

Each abstract assertion is a symbolic expression of the form *Antecedent* $\stackrel{\text{LEADSTO}}{\Longrightarrow}$ *Consequent*. The *Antecedent* specifies the current state of the abstract machine and the current inputs. The *Consequent* specifies what the outputs and new state of the machine should be, after the abstract machine makes a transition.

For example, the abstract assertion for a *read* operation at port 0 of the register file (see Section 3.1) is

$$(Op = Read) \wedge (R[i] = u) \wedge (TR[n] = v) \wedge \quad (1)$$

$$(ReadAdr0 = j) \wedge (TGPRread = 0) \wedge (ReadEn0 = 1) \quad (2)$$

$$\stackrel{\text{LEADSTO}}{\Longrightarrow}$$

$$(R[i] = u) \wedge (TR[n] = v) \wedge \quad (3)$$

$$(when(i = j)(Dout0 = u)) \quad (4)$$

The antecedent says that a read operation is being performed, register R_i contains the symbolic value u, register TR_n contains the value v (line 1), the read address is the symbolic value j, the first register bank is being read, and reading is enabled (line 2).

In the consequent, line 3 states that the register values remain unchanged in the read operation e.g. if register R_i contains the value u initially, after a read operation at some address j, R_i still contains u. In line 4, we verify that the read operation results in the correct data value being sent to the output. When R_i contains u, and the read address is j, we are interested in checking the output for data u, only when $(i = j)$, which is expressed by the $when$ condition. Using symbolic values i, j, u, n, and v allows us to cover all pertinent cases with such an assertion.

The implementation mapping, illustrated in figure 3, expands such an assertion to include details of the circuit implementation, such as the timing of state, IO, and clock signals. It maps the value of each component of the abstract state (for example, $ReadAdr0$) onto the values of one or more specific circuit nodes at specific times. For example, the figure shows how $(op = Read)$ in the abstract assertion translates to the READ_CLK signal making low to high and then high to low transitions during specified times, and the WRITE_CLK signal staying low.

Note that the variables i, n, and j in the assertion are used as array indices. The implementation mapping represents each of them in binary form as a word of symbolic Boolean variables. From the antecedent fragment $R[i] = u$, the implementation mapping will initialize each RAM storage node s in the register file with a symbolic ternary function

$$f_s(i, u) = \begin{cases} u, & \text{if } i = s \\ X, & \text{otherwise} \end{cases}$$

where X is the ternary constant of switch-level simulation. This technique, called *symbolic indexing* [3], is critical to the efficiency of STE on arrays [1, pp. 161–163]. It is responsible for reducing the number of variables STE considers to a number approximately logarithmic in the number of array locations.

5. State node identification

In order to apply the methodology above, it is necessary to expose the internal circuit state to the implementation mapping. In the register file, for instance, the internal state consists of the registers in both register banks. In the DTAG circuit, the states which we need to expose are all the tag, valid, modified, and LRU bits.

We worked with flat transistor netlists in which it was not immediately obvious if a given node was a memory storage node, and if so which location it represented. To address this, we created an automated state node identification method [8] which identifies the storage nodes in a SRAM array. This method does a write operation with symbolic data and address, resulting in a unique symbolic indexing function being exhibited on each memory storage node. Searching the circuit nodes for these ternary functions yields all the storage nodes and the memory locations they represent. This technique made it possible to identify the state nodes in the register file and the tag, valid and modified bits in the DTAG circuit.

Since the LRU state nodes can not be written to directly by using the regular DTAG operations, the above technique needed extension. The LRU bits can be reset to all zeros or they can change as a result of a tag operation to reflect the past memory access pattern. We made use of this property to put unique symbolic values on the LRU nodes by symbolic simulation. The LRU bits were first reset, and then we performed a two DTAG operations such that a symbolic way w (encoded with Boolean variables w_1 and w_0) was accessed in the first operation, and symbolic way v (encoded with Boolean variables v_1 and

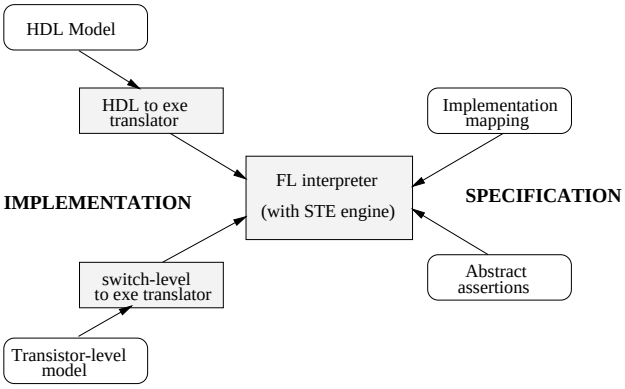

Figure 4: Tool organization for our verification experiments.

v_0) was accessed in the second operation. This put unique symbolic Boolean values (Boolean functions of w_1, w_0, v_1 and v_0) on all the six LRU bits of a set. From our knowledge of how the LRU bits get updated when a DTAG operation occurs, we were able to identify all the LRU bits of all the sets.

6. Experiments and results

In the discussion that follows, all assertions will be written in a form of FL pseudo-code. This pseudo-code gives the flavor of FL, while being more readable.

6.1. Tool organization

The FL interpreter shown in figure 4 includes a STE engine which can accept STE assertions. As described in section 4, we structured our specification as abstract assertions and implementation mapping, both described in the FL language. This specification was converted into STE assertions by the interpreter. The interpreter accepts hardware designs in the $.exe$ format [11], which essentially describes the hardware as a set of excitation functions for the circuit nodes. We converted our transistor-level designs to the $.exe$ format by using a translator included with the Voss system. An internal HDL is used to describe hardware design at the RTL level. In order to run STE on the RTL designs, we built a HDL to $.exe$ translator.

6.2. Creation of a switch-level model

A prerequisite to running STE on a transistor circuit is an accurate switch-level model. The default transistor strengths and node sizes used in our circuit design methodology are often sufficient to run switch-level simulation, but not for RAM core sense amps and some precharged circuits.

To run Voss on our circuits we had to translate our schematics into a flat netlist format. For the register file circuit, no manual intervention was necessary to create an accurate switch-level model. The DTAG circuit, however, having sense amps and non trivial internal timing, including self-timed components, required some hand modeling. Voss allows back-annotating circuit nodes with delay values, and we found this very useful for creating the DTAG switch-level model. We also religiously avoided the use of transistor directions, preferring instead to model current paths by increasing the number of strength levels. We avoid transistor directions because they can mask real circuit behavior, and consequently, real circuit bugs. It was necessary, in places, to increase the number of transistor strength levels by hand, to solve modeling problems.

6.3. Register file

To verify the register file, we wrote six assertions. Five describe the read operation at each of the five read ports, and the sixth describes the register-file write operation. The assertions to verify the read operation are similar to the one described in section 4. Each assertion also includes the enable signal $ReadEn0$, and verifies that the outputs have the correct value for both high and low values of this signal. Details of the write operation assertions appear below. The implementation mapping is very similar to the illustration of Figure 3.

6.3.1. Write operation

The register file has two write ports which can update the registers in parallel. The assertion describing the write operation appears below. It shows a subset of the various possible combination of control signals for write.

$$(R[i] = u) \wedge (TR[n] = v) \wedge \quad (5)$$
$$(writePort0(j, x, wen0)) \wedge (writePort1(k, w, wen1)) \wedge \quad (6)$$
$$(TGPRwrite = twr) \wedge (writeInhibit = winhibit) \quad (7)$$
$$\stackrel{\text{LEADSTO}}{\Longrightarrow}$$
$$(when(writeNone)(R[i] = u)) \wedge \quad (8)$$
$$(when(writeNone)(TR[n] = v)) \wedge \quad (9)$$
$$(when(twr)(R[i] = u)) \wedge \quad (10)$$
$$(when(\neg twr)(TR[n] = v)) \wedge \quad (11)$$
$$\ldots$$
$$(when(i \neq j \wedge i \neq k \wedge write01\_gpr)(R[i] = u)) \wedge \quad (12)$$
$$(when(i = j \wedge i \neq k \wedge write01\_gpr)(R[i] = x)) \wedge \quad (13)$$
$$(when(i \neq j \wedge i = k \wedge write01\_gpr)(R[i] = w)) \quad (14)$$

$$writeNone = \neg(wen0 \vee wen1) \quad (15)$$
$$write01gpr = \neg winhibit \wedge wen0 \wedge wen1 \wedge \neg twr \quad (16)$$

Lines 5 through 7 are the antecedent. Line 5 states that initially register R_i contains the symbolic value u and register TR_n contains the symbolic value v. A write operation is done at write port 0 with symbolic address j, symbolic data x and the write enable for the port set to symbolic value $wen0$ (line 6). Similarly, a write is done at write port 1, with address k, data w and write enable $wen1$. Finally, the symbolic value twr controls which of the two banks the writes go to and, $winhibit$ when true inhibits all writes.

To make the consequent more readable, we have used the abbreviations $writeNone$ and $write\_01gpr$. These have been described in terms of the symbolic variables in lines 15 and 16. $writeNone$ describes the condition that writes to both ports are disabled. $write01\_gpr$ describes the condition when writes through both ports are enabled, and the writes go to the first bank of the register file.

The consequent consists of lines 8 through 14. Lines 8 and 9 express the condition that when both write ports are disabled, all the registers remain unchanged. When writes are done to the second bank of registers, the first bank remains unchanged (line 10), and vice-versa (line 11). When register i contains the value u initially, and write addresses at both ports do not match i, then the value of register i remains unchanged (line 12). If i matches the address at the first write port ($i = j$), but not the second port ($i \neq k$), then register i gets updated to the data at the first port (line 13). Since, it is specified that

write addresses at the two ports will not be the same, we do not check the results of write when the write addresses match, i.e. $i = j = k$.

6.3.2. Resource requirements

The verification of the register file takes a total of 267 seconds (on a IBM RS6000/580) for the complete set of assertions and generates a maximum of 8875 OBDD nodes. Voss used 31 MB of memory; 21 MB was used to represent the circuit nodes, and the excitation functions, and the rest was taken up by OBDDs and other run-time structures created by the FL interpreter.

6.4. Data cache tags

The data cache tags unit can do the following operations on any clock cycle – reset, load request, store request, snoop kill, snoop flush, tag refill, and status write. The assertions for some of these operations are described below.

6.4.1. Reset operation

The reset operation resets the tags unit by zeroing all the valid, modified and LRU bits. This can be succinctly expressed by the following assertion:

$$(op = reset) \overset{\text{LEADSTO}}{\Longrightarrow} (V[i][w] = 0) \wedge (M[i][w] = 0) \wedge (L[i] = 0x00)$$

6.4.2. Tag write operation

In this operation, tag bits and valid and modified (status) bits are written to a given way of a set. As a result of the write the LRU bits get updated to make the written way the most recent way. This operation can be specified by the assertion below.

$$\begin{aligned}
&(op = tagwrite) \wedge \\
&(T[i][w] = t) \wedge (V[i][w] = v) \wedge && (17)\\
&(M[i][w] = m) \wedge (L[i] = l) \wedge && (18)\\
&(writeindex = wi) \wedge (writeway = ww) \wedge && (19)\\
&(writetag = wtag) \wedge (writestatus = wstat) && (20)\\
&\qquad\qquad \overset{\text{LEADSTO}}{\Longrightarrow} \\
&(when(wi \neq i \vee ww \neq w) \\
&\quad ((T[i][w] = t) \wedge (V[i][w] = v) \wedge (M[i][w] = m))) \wedge && (21)\\
&(when(wi = i \wedge ww = w) \\
&\quad ((T[i][w] = wtag) \wedge (V[i][w] = wstat[0]) \wedge \\
&\qquad\qquad (M[i][w] = wstat[1]))) \wedge && (22)\\
&(when(wi \neq i)(L[i] = l)) \wedge && (23)\\
&(when(wi = i)(L[i] = update(l, ww))) && (24)
\end{aligned}$$

In the antecedent, lines 17 and 18 show the initial system state. They state that the tag value, valid bit and modified bit of the ith set and the wth way are t, v and m respectively. It also states that initially the LRU bits of the ith set is l. The next two lines show that tag value $wtag$ and the status bits $wstat$ are written to set wi and way ww.

As a result of the tag write, the tag, valid and modified bits of the addressed way get updated and all other ways remain unchanged. This is shown in lines 21 and 22. Line 24 shows that for an addressed set, the LRU bits get updated to reflect access to way ww, and they remain unchanged for a set that is not addressed (line 23).

The status write operation is very similar to the tag write operation – the only difference is that tags bits are not written during a status write.

6.4.3. Load request operation

For verifying the load request operation we wrote two assertions. The first assertion shows that if the initial machine state is
$(T[i][0] = t0) \wedge (T[i][1] = t1) \wedge (T[i][2] = t2) \wedge (T[i][3] = t3) \wedge$
$(V[i][0] = v0) \wedge (V[i][1] = v1) \wedge (V[i][2] = v2) \wedge (V[i][3] = v3) \wedge$
$(M[i][0] = m0) \wedge (M[i][1] = m1) \wedge (M[i][2] = m2) \wedge$
$(M[i][3] = m3) \wedge (LRU[i] = l)$,
and a load request is done with an index value of i and the tag input is $tagin$, then one of the following two things happen:

1. One of the four ways hit:
   For example, way 0 hits when $t0 = tagin$, and the valid bit for way 0, $v0$, is true. The LRU bits get updated to reflect that way 0 was most recently accessed, and all other state bits remain unchanged. The HIT output becomes true, HITWAY becomes 00 to reflect that way 0 has been hit, and at the dirty bit output, the value of $m0$, the dirty bit of way 0, is written out.

2. None of the ways hit: In this case all the state bits remain unchanged, and the dirty and the tag bits of the way to be replaced (least recently used way) are written out.

Certain combinations of state bit values are forbidden in this circuit. For instance, in a set no two tags can match, and it is assumed that the environment always maintains this state invariant by not writing in matching tag values Similarly, only certain combinations in the LRU bits are legal, and only these represent valid LRU information. All the DTAG actions above have been verified under these invariant conditions. A second assertion verifies that the tag, valid, modified and the LRU bits for a set that is not indexed remain unchanged in a load operation. Store request and snoop operations have not been described here, but they are very similar to the load request operation.

6.4.4. Resource requirements

The verification of the DTAG circuit takes about 10 minutes (on a IBM RS 6000/580) for the most complex of the assertions (e.g., the store request assertion) and generates 110,000 OBDD nodes. Voss used 150 MB of memory (of which 103 MB is to represent the circuit).

6.5. Bugs

In the process of verification, no bugs were found in the actual, register-file circuit. The designer did, however, test our methods by making two copies of the design, inserting a bug into one copy, and seeing if our tool could find it (it did). In addition, we translated and ran Voss on the RTL version of the register file, and found that it did not obey the specification. The "misbehavior", however, was in an underspecified area: when addressing a register in TR0-TR3, the specification states that the two most significant address bits were don't cares. However, the simulation model went into an error state if 1's were asserted on these lines, and this was detected by a failure of our assertion, in Voss. The transistor netlist under the same conditions, completed the write (and the same assertion passed). This difference was detected, and showed the power of STE methods. It did not affect correct modeling of the register file inside the larger chip, however,

since the surrounding circuitry to the register file did, in fact, keep these bits low during writes to TR0-TR3.

Two actual bugs were discovered in the DTAG circuit. The first bug, a serious functional error, was known beforehand, but its nature was kept secret from the person running the STE verification. This bug was due to a transistor "sneak path" (i.e., a signal running in an unexpected direction) in the "hit" detection circuitry of the DTAG. This error was masked in regular verification process because of the assignment of directions to transistors. In addition, it is not clear if the appropriate digital vector would have been found to reveal it, had the directions not been applied. A single symbolic simulation vector, used during creation of the switch-level model, brought out this bug. This bug had already been fixed, in a later version of the circuit than the version upon which we were working.

The second bug was discovered when an assertion for what is called the status-write operation failed. Tracing the cause of the failure revealed that the LRU bits had not been updated, contrary to the specification. The LRU bits determine which line in a cache set will be replaced, in the event the set becomes full and a new line must be brought in. Faulty LRU bits merely cause discrepancies in performance (the line replaced may be the one most needed, for instance). This makes bugs in LRU bits difficult to find in digital simulation, unless one specifically monitors these bits on a cycle by cycle basis.

6.6. Debugging

In our verification effort, we needed switch-level debugging capabilities for two separate tasks, initially to create the switch-level model of the DTAG circuit, and later to trace the causes of assertion failures. Voss proved to be a good tool for both tasks.

For creating a switch-level model, we used Voss as a symbolic simulator, with assertions that had the circuit stimulus as the antecedent and an empty consequent. Using a mix of symbolic and constant stimulus values proved valuable in tracing signals and determining which part of a circuit was not correctly modeled. Symbolic ternary functions which appear on circuit nodes give a wealth of information on circuit operation because they represent the result of many different simulation cases runs at once.

When an assertion fails, STE returns a Boolean function of symbolic variables in the assertion which indicates the reason for the failure. Any assignment to the symbolic variables which makes the Boolean function false is a counter- example. We used this information to substitute constants for some of these variables in these failing assertions. With fewer symbolic variables, it is easier to understand symbolic values that appear on circuit nodes, simplifying the task of tracking down problems.

7. Conclusion

We have found STE to be a powerful method for verifying on-chip memory arrays, and Voss to be a flexible tool. Our results are encouraging and have reinforced our belief that the logic of STE is sufficient for specifying properties of data-intensive systems. We have verified circuits with a rigor not possible by conventional simulation. We detected design errors and specification ambiguities that were non-obvious, and traced their cause with ease—all with a remarkably small expenditure of memory storage and CPU time.

An important side effect of this effort is that the assertions now serve as design documentation. The HDL description, often claimed as the circuit specification, is not really a specification. It is an implementation of that specification. The assertions generated for using STE come much closer to capturing the design intent.

We look forward to working, in the future, on fashioning STE into an easy to use tool for arrays. We anticipate working on a simplified, array-specific user interface, and on better automation for switch level modeling.

8. Acknowledgments

Our thanks go to Carl Seger, who answered our questions on Voss and added some features for us, to Brian Branson, Paul Reed, Mike Brauer and Cody Croxton, for helping us understand the circuits we verified, and to Scott Butler, Charlie Malley and Hemendra Talesra who helped define sensible goals for our verification effort.

References

[1] D. L. Beatty, "A Methodology for Formal Hardware Verification with Application to Microprocessors," Ph.D. Thesis, published as Technical report CMU–CS–93–190, School of Computer Science, Carnegie Mellon University, August 1993.

[2] D. L. Beatty and R.E. Bryant, "Formally verifying a microprocessor using a simulation methodology," DAC, 1994.

[3] D. L. Beatty, R.E. Bryant, C. J. H. Seger, "Synchronous circuit verification by symbolic simulation: an illustration," Advanced Research in VLSI: Proceedings of the 6th MIT Conference, pp. 98–112, MIT Press, March 1990.

[4] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," IEEE Transactions on Computers, C-35(8), August, 1986

[5] R. E. Bryant, "Symbolic simulation—techniques and applications," DAC, 1990.

[6] E. M. Clarke, E. A. Emerson, A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," ACM Transactions on Programming Languages and Systems, 8(2):244–263, 1986.

[7] A. Kuehlmann, A. Srinivasan, D. P. LaPotin, "Verity—a formal verification program for custom CMOS circuits," IBM Journal of Research and Development, 39(1/2), January/March 1995.

[8] M. Pandey, R. E. Bryant, "Memory array state node identification tool," *accepted for publication in* Motorola Technical Developments, Motorola Inc.

[9] L. C. Paulson. *ML for the Working Programmer*. Cambridge Univ. Press, 1991.

[10] C. J. H. Seger, R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," Formal Methods in System Design,6:147–189 (1995).

[11] C. J. H. Seger, "Voss—a formal hardware verification system: user's guide," Technical Report 93-45, Department of Computer Science, University of British Columbia, 1993.