Formal Verification of Content Addressable Memories using Symbolic Trajectory Evaluation *

Manish Pandey[1] Richard Raimi[2] Randal E. Bryant[1] Magdy S. Abadir[2]

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

Abstract

In this paper we report on new techniques for verifying content addressable memories (CAMs), and demonstrate that these techniques work well for large industrial designs. It was shown in [6], that the formal verification technique of symbolic trajectory evaluation (STE) could be used successfully on memory arrays. We have extended that work to verify what are perhaps the most combinatorially difficult class of memory arrays, CAMs. We use new Boolean encodings to verify CAMs, and show that these techniques scale well, in that space requirements increase linearly, or sub-linearly, with the various CAM size parameters.

In this paper, we describe the verification of two CAMs from a recent PowerPC™ microprocessor design, a Block Address Translation unit (BAT), and a Branch Target Address Cache unit (BTAC). The BAT is a complex CAM, with variable length bit masks. The BTAC is a 64-entry, 64-bits per entry, fully associative CAM and is part of the speculative instruction fetch mechanism of the microprocessor. We believe that ours is the first work on formally verifying CAMs, and we believe our techniques make it feasible to efficiently verify the variety of CAMs found on modern processors.

1. Introduction

Content Addressable Memories (CAMs) play an important role in many modern digital systems. CAMs are widely used wherever fast parallel search operations are required. Some examples of CAMs found on modern processors are translation-lookaside buffers (TLBs), branch prediction buffers, branch target buffers and cache tags. CAMs have also been used in such applications as data compression, data-base accelerators, and network routers.

CAMs in microprocessors are usually custom designed at the transistor level, as these circuits are often in the critical path of a chip and it is necessary to optimize area and performance. These circuits often include self timed components and other complex forms of circuitry, and they typically have complex internal timing. *For these reasons it becomes necessary to verify such designs at the transistor level.* All the designs in this paper have been verified at the transistor level using a switch-level model.

* 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 by a grant from Motorola.

0

Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and /or a fee.
DAC 97, Anaheim, California
© 1997 ACM 0-89791-920-3/97/06 ..$3.50

While work on verification of memory arrays has been reported in [2], [6], and [3], there has been little published on the particular needs of CAMs. In [2, pp. 102], Bryant comments on the difficulty of CAM verification as follows: "*...Other classes of memory designs can also be verified by simulating a linear, or nearly-linear number of patterns. ... On the other hand, content-addressable memories do not seem to fit into this class, since it is not easy to identify where a particular datum will be stored.*"

In this paper, we describe how we leveraged the formal verification technique of symbolic trajectory evaluation (STE), along with new Boolean encoding techniques, to verify CAMs. The new encodings were needed to contain the exponential growth in the space requirements with increasing CAM sizes, which could occur with a naive use of variables in STE. Our work shows that we were able to solve this problem and formally verify these types of circuits, with very modest space and time requirements.

In the remainder of this paper we discuss background material on STE and CAMs (Section 2), and we then describe experiments done on small, generic CAMs in which we perfected the needed Boolean encodings (Section 3). Finally, we describe how we used our techniques, with success, on two complex CAMs (Sections 4 and 5) from a recent PowerPC processor. This verification was carried out at the joint Motorola-IBM PowerPC design center, Somerset, located in Austin, Texas. In our work, we utilized the Voss STE system [8].

2. Background

2.1. Symbolic trajectory evaluation

STE [7] is a ternary symbolic simulation based technique for verifying behaviors of a system over bounded, finite time intervals. Specifications are *trajectory assertions* of the form $[Antecedent \Rightarrow Consequent]$, where $Antecedent$ and $Consequent$ are *trajectory formulae*. Intuitively, the antecedent defines an initial setting and a stimulus pattern for the circuit nodes, while the consequent defines the expected response.

The basic element of a trajectory formula (TF) is a simple predicate, e.g., ($node_i$ is 0), which states that $node_i$ of a circuit contains the value 0 at the present time. Using conjunction, case restriction and a next-time operator, trajectory formulas can be constructed from the simple predicates. If G_1 and G_2 are TFs, then their conjunction $G_1 \wedge G_2$ is also a TF. If G is a TF, then $\mathcal{N}G$ is a TF, where $\mathcal{N}$ is the next-time temporal operator, and $\mathcal{N}G$ means that G holds in the next time step. Finally, if G is a TF, and E is a Boolean expression, then $(when\ (E)\ G)$ is a TF, where $when$ is the domain restriction operator. $(when\ (E)\ G)$ means that G must hold when E is true. This simple logic is sufficient to verify the class of systems, such as arrays, for which functionality can be partitioned into a set of operations each of which update the system

state in a deterministic manner. Given an assertion $[A \Rightarrow C]$, the circuit is simulated with ternary symbolic simulation patterns derived from the antecedent A. During simulation, at each time step the circuit state is checked against the expected response specified in C. The ternary symbolic values generated in the process are represented by pairs of Ordered Binary Decision Diagrams (OBDDs).

The temporal logic of STE is quite restricted as compared to the temporal logics of model checkers like SMV [4]. However, the use of such a logic obviates the need to represent a system's transition relation and calculate its reachable state set, two very expensive operations. Because of this, STE excels in the verification of large, data intensive systems, such as memory arrays, having tens of thousands of state holding elements.

Recently a methodology for application of STE was developed [1] and we have adapted this for verification of CAMs. In this methodology, the desired system behavior is specified as a set of assertions over *abstracted system space*. Each abstract assertion, which is of the form $[A \overset{\text{LEADSTO}}{\Longrightarrow} C]$ describes how the system operations transform the abstract system space. Intuitively, the abstract assertion's antecedent, A, specifies the current abstract state and inputs, and the consequent, C, specifies the abstract state and outputs after a system operation. In addition to the abstract assertions, the user provides an *implementation mapping*, giving the correspondence of the abstract state to the nodes in the transistor-level circuit, and giving the timing of signal transitions. Given this mapping, the abstract specification is mapped into a set of trajectory assertions, to be checked by the STE decision procedure. For details on this methodology, the reader is referred to Beatty's thesis [1].

The abstract assertions shown in this paper use a pseudo-code notation. The actual assertions were written in the FL language, which is the functional language front-end to the Voss verifier.

2.2. An Illustration of the Methodology

To illustrate the methodology described above, consider the abstract assertion below. It describes a tag write operation for the CAM circuit in section 2.3.

$$(op = tag_write) \wedge (Tadr = adr) \wedge (Tagin = newtag) \wedge (T[i] = tag) \overset{\text{LEADSTO}}{\Longrightarrow}$$

$$(when(i \neq adr)(T[i] = tag)) \wedge (when(i = adr)(T[i] = newtag))$$

The antecedent (from left to right) states that a tag write operation is being done to address adr with value $newtag$, and in the initial state of the verification, the ith tag entry contains the symbolic value tag. The consequent specifies that the addressed tag entry is changed to *newtag*, and all other tag entries are unchanged.

The implementation mapping relates the abstract state of each "phrase" in the abstract assertion to concrete signal timing and valuations on actual circuit nodes. It captures implementation details such as that two phase clocking is used, and that the $Twrite$ signal is asserted, and the tag data and addresses provided, with appropriate setup and hold times, when $Ph1$ is high. For an example of such details, see [6, pp. 651].

Note that the variables i and j in the abstract assertion above are used as array indices. The implementation mapping represents each of them in binary form as a vector of symbolic Boolean variables. From the phrase, $(T[i] = tag)$, in the antecedent of the abstract assertion, the implementation mapping will initialize each tag storage node k in

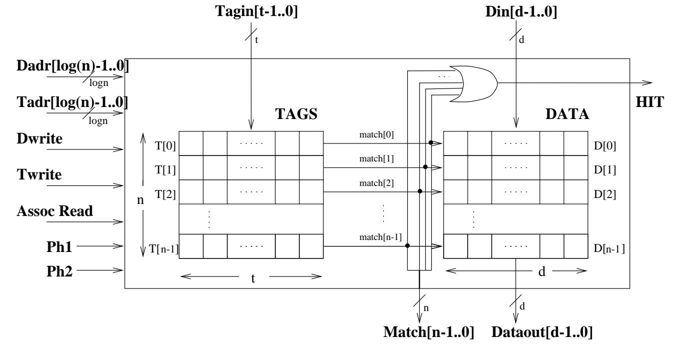

Figure 1: Content Addressable Memory: Tag size = t, Number of entries = n, Data size = d

the CAM with a symbolic ternary function, $f_k(i, tag)$, which returns tag, when $i = k$, and X otherwise (X represents the "unknown" or "don't care" value of switch-level simulation). This technique, called *symbolic indexing*, is critical to the efficiency of STE on memory-based circuits[1]. It is responsible for reducing the number of variables in an STE verification to a number logarithmic in the number of array locations.

2.3. A CAM design

Generally, CAMs employ as an identifier a bit field called a *tag*, each tag identifying a particular data entry stored in the array. CAMs vary depending upon data and tag size, techniques to read and write contents and mark contents as valid, tag masking fields, etc. In spite of all this diversity, CAMs all have in common the *associative read capability*[1]. The associative read operation consists of searching, in parallel, all tags in the CAM to determine if there is a match to a particular tag of interest, and then sending the associated data entry to an appropriate read port of the memory.

The high-level design shown in Figure 1 is a very basic CAM. We implemented this design as an experimental vehicle for finding better Boolean encodings for CAM verification (see Section 3). This design has n tag entries, $T[0], T[1], ..., T[n-1]$. Corresponding to each tag entry, $T[i]$, there is a data entry $D[i]$. The most distinctive operation of this circuit is the *associative read* operation. In this operation $Tagin$ is compared in parallel with all the $T[]$ tag entries, and if there is a match on the ith tag entry, then HIT rises, and $D[i]$ appears at $Dataout$. If there is no match to $Tagin$, i.e., a *miss*, HIT remains low (and, the surrounding circuitry would ignore $Dataout$). We have implemented this design as a transistor-level netlist using a tool called cmu-netlist. Each n-bit tag consists of n tag cells. Each tag cell contains 9 transistors and its design is based on the one in [9, pp.590].

It is an assumption that, among valid tag entries there would be *at most one* tag that would match $Tagin$. This property, the *at most one tag match property* is an important system invariant. However, it is usually not enforced in hardware. Rather, CAMs generally depend upon surrounding circuitry, or the software manipulating the entire chip, to maintain this invariant. For example in the PowerPC BAT array [5], the responsibility of maintaining the invariant is with the operating system. In Section 3, we show how we used this invariant to efficiently verify a CAM. Even when the circuit design is enhanced to handle *multiple matches*, we can verify CAMs efficiently using techniques

[1] In some instances CAMs also have an *associative write capability*. The PowerPC Branch Target Address Cache circuit is one such example and it is described in Section 4.

based on those outlined here. However, due to space limitations, we will not discuss that in this paper.

3. CAM properties and CAM encodings

Below we show how a well chosen encoding can dramatically reduce the number of variables, and therefore the number of OBDD nodes, required for the verification.

3.1. CAM Encodings

We will discuss the CAM encoding problem in the context of verifying the associative read operation of CAMs. We will refer to a generic CAM modeled after that of Figure 1, in Section 2.3.

The most obvious approach to verifying the associative read operation is to introduce a Boolean variable for each bit of state in the $T[i]$ and $D[i]$ tag and data entries. We illustrate this below with an example trajectory assertion. Assume the number of CAM entries, n, equals 3. Let $\vec{tin}$, $\vec{t_0}$, $\vec{t_1}$ and $\vec{t_2}$ be vectors of Boolean variables of size t, the width of the $T[i]$ entries. Let $\vec{d_0}$, $\vec{d_1}$ and $\vec{d_2}$ be vectors of Boolean variables of size d, the width of the $D[i]$ entries. The following assertion specifies the associative read operation under these conditions[2].

$$(op = assocread) \wedge (Tagin = \vec{tin}) \wedge$$
$$(T[0] = \vec{t_0}) \wedge (T[1] = \vec{t_1}) \wedge (T[2] = \vec{t_2}) \wedge$$
$$(D[0] = \vec{d_0}) \wedge (D[1] = \vec{d_1}) \wedge (D[2] = \vec{d_2})$$
$$\stackrel{\text{LEADSTO}}{\Longrightarrow}$$
$$(when(nomatch)((HIT = 0))) \wedge$$
$$(when(matchonly0)((HIT = 1) \wedge (Dataout = \vec{d_0})) \wedge$$
$$(when(matchonly1)((HIT = 1) \wedge (Dataout = \vec{d_1}))) \wedge$$
$$(when(matchonly2)((HIT = 1) \wedge (Dataout = \vec{d_2})))$$

The first line of the antecedent specifies that an associative read is being done and the input data is $\vec{tin}$. The second line specifies that the three tag registers initially contain $\vec{t_0}$, $\vec{t_1}$, and $\vec{t_2}$. The three data registers are specified as initially containing $\vec{d_0}$, $\vec{d_1}$, and $\vec{d_2}$. To simplify the consequent, we use the following Boolean functions, $match0 = (\vec{tin} = \vec{t_0})$, $match1 = (\vec{tin} = \vec{t_1})$, $match2 = (\vec{tin} = \vec{t_2})$, $nomatch = \neg(match0 \vee match1 \vee match2)$, $matchonly0 = match0 \wedge \neg match1 \wedge \neg match2$, $matchonly1 = \neg match0 \wedge match1 \wedge \neg match2$, and $matchonly2 = \neg match0 \wedge \neg match1 \wedge match2$. The first line in the consequent checks that there are no matching entries in the CAM. The second line checks for HIT and $Dataout$ when only the first entry matches. Note that we do not check for conditions inconsistent with the *at most one tag match* system invariant. For example, we do not check for what happens if $(\vec{tin} = \vec{t_0})$ and $(\vec{tin} = \vec{t_1})$ are both true. A total of $(t + d)n + t$ Boolean variables are needed for this assertion. We call this encoding, where every circuit state bit has a corresponding Boolean variable, the *full encoding*.

We can reduce the variable count, however, by using symbolic indexing. At this point we will use it just for the data entries. To effect this, the antecedent should be changed to contain $(D[\vec{i}] = \vec{data})$ instead of $(D[0] = \vec{d_0}) \wedge (D[1] = \vec{d_1}) \wedge (D[2] = \vec{d_2})$. $\vec{data}$ is a vector of Boolean variables d bits wide, and $\vec{i}$ is a vector of Boolean variables $\lceil log_2 n \rceil$ bits wide. The consequent is also changed. Line 2 of the consequent is changed to (lines 3 and 4 are changed similarly):

[2] Some parts of the assertion necessary for verification thoroughness, e.g. that the tag and data bits are unchanged on a read, have been omitted.

$(when(match0 \wedge \neg match1 \wedge \neg match2 \wedge (\vec{i} = 0))((HIT = 1) \wedge (Dataout = \vec{data}))$.

This encoding needs only $(n+1) \cdot t + d + \lceil log_2 n \rceil$ Boolean variables. We call this the *plain encoding*. For identical data and tag sizes, the number of variables goes down by half, as compared to the *full encoding*. However, as later results will show, with increasing n, memory requirements can still grow rapidly with the plain encoding scheme. So we must improve on it.

We can reduce the number of variables further, by taking advantage of the *at most one tag match* system invariant. Let $Tagin$ be $\vec{tin} = \langle tin_{t-1}, tin_{t-2}, ..., tin_0 \rangle$. In order that the tag entry $T[0]$ not match $\vec{tin}$, it should be one of the following t ternary vectors: $\langle \neg tin_{t-1}, X, ..., X \rangle$, $\langle X, \neg tin_{t-2}, X, ..., X \rangle$, ..., $\langle X, X, ..., X, \neg tin_0 \rangle$. The position at which the tag in $T[0]$ is unequal can be encoded by, p, a vector of $\lceil log_2 t \rceil$ variables. So the condition that $T[0]$ is not equal to $\vec{tin}$ can be written as

$$(when(p = 0)(T[0][0] = \neg tin_0)) \wedge$$
$$(when(p = 1)(T[0][1] = \neg tin_1)) \wedge$$
$$... \wedge$$
$$(when(p = (t-1))(T[0][t-1] = \neg tin_{t-1}))$$

We abbreviate this as $\exists \vec{p}. T[0][\vec{p}] = \neg tin_{\vec{p}}$, i.e. there exists a $\vec{p}$ such that at the pth bit position there is a mismatch between $\vec{tin}$ and the tag entry $T[0]$.

We now verify the associative read operation in two parts. First, we verify the case where no CAM entries match the input tag, and then we verify the case where the ith entry does match the input tag. For the case where no hit occurs the new assertion is:

$$(op = assocread) \wedge (Tagin = \vec{tin}) \wedge$$
$$(\exists p_1. T[0][p_1] = \neg tin_{p_1}) \wedge$$
$$(\exists p_2. T[1][p_2] = \neg tin_{p_2}) \wedge$$
$$(\exists p_3. T[2][p_3] = \neg tin_{p_3})$$
$$\stackrel{\text{LEADSTO}}{\Longrightarrow}$$
$$(HIT = 0)$$

where p_1, p_2 and p_3 are encoded by vectors of variables, indicating the position at which the mismatch with $\vec{tin}$ occurs. For the case where one entry matches the input tag, we write:

$$(op = assocread) \wedge (datain = \vec{tin}) \wedge (D[i] = \vec{data}) \wedge$$
$$(when(i = 0)(T[0] = \vec{tin})) \wedge (when(i \neq 0)(\exists p_1. T[0][p_1] = \neg tin_{p_1})) \wedge$$
$$(when(i = 1)(T[1] = \vec{tin})) \wedge (when(i \neq 1)(\exists p_2. T[1][p_2] = \neg tin_{p_2})) \wedge$$
$$(when(i = 2)(T[2] = \vec{tin})) \wedge (when(i \neq 2)(\exists p_3. T[2][p_3] = \neg tin_{p_3}))$$
$$\stackrel{\text{LEADSTO}}{\Longrightarrow}$$
$$(HIT = 1) \wedge (match[k] = (k = i)) \wedge (output = data)$$

This encoding requires only $(log_2 n + n \cdot log_2 t + t + d)$ Boolean variables, a substantial savings over the two earlier encodings. We refer to this encoding as the *CAM encoding*. As will be seen in section 3.3, verification of even moderate sized CAMs would be intractable without an encoding at least as efficient as the CAM encoding.

It is instructive to compare the number of Boolean variables required for the three different encodings, with the number required for representing a transition relation. For a 16 entry CAM, with 16 bit tag and data sizes, the number of Boolean variables required for the full, plain and CAM encodings are 528, 292, and 100 respectively. For the transition relation the required number of Boolean variables is over 1024, which is double the number of state elements.

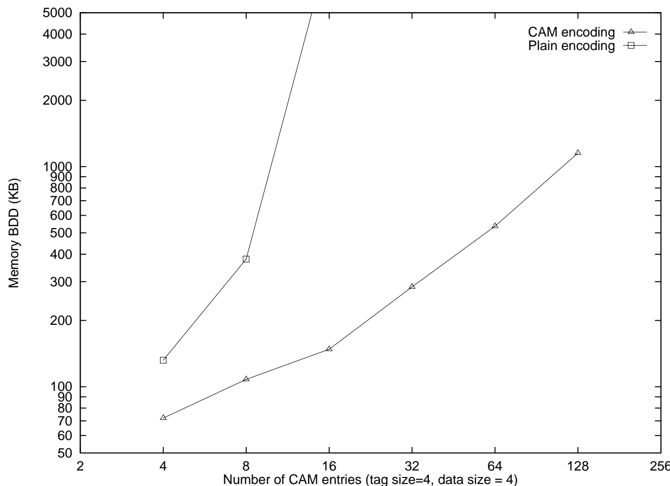

Figure 2: CAM: number of tag entries vs. OBDD sizes

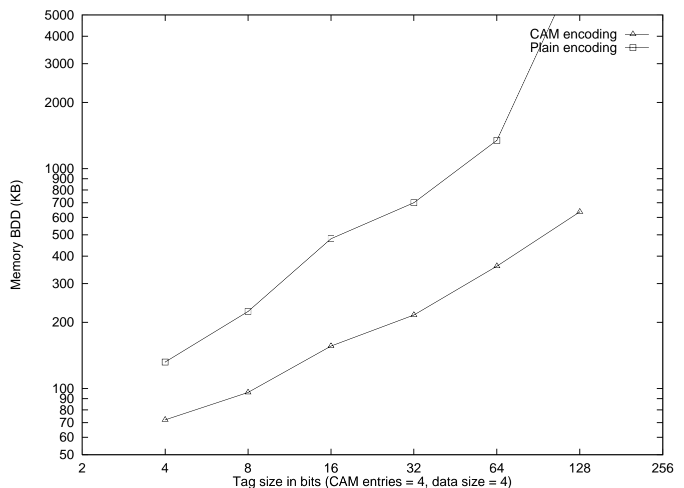

Figure 3: OBDD trends with varying tag size.

3.2. Experimental Results and Discussion

In Figures 2 and 3 we have plotted the results from verification of different size CAMs, using the CAM encoding and the plain encoding. The full encoding is not included here, as it usually performs much worse than the other two encodings. We have plotted the memory taken by the OBDDs generated when verifying the associative read operation. All other verified CAM operations take less space, and have not been included here. The OBDD variable ordering for the experiments was carefully chosen, so that, as much as possible, we would avoid unfair comparisons between the two encodings. For each encoding, we chose an initial variable ordering that, from our understanding of the circuit function, would result in small OBDDs. Upon running STE with the initial variable ordering, the OBDD package dynamically reordered some of the variables. We used this reordering information to improve our understanding of the variable interaction and further tuned the variable ordering to minimize the OBDD sizes before running STE again.

Figure 2 shows how the OBDD sizes for the plain and CAM encoding vary for CAMs with varying associativities (tag and data sizes are constant). As the graph shows, there is a dramatic difference in the space taken by the two encoding approaches. As the number of tag entries increases, the plain encoding requires substantially more memory than the CAM encoding. Many TLBs are highly associative, and for such circuits the plain encoding approach will clearly not work. These results motivated us to use CAM encodings in all our further CAM verifications (Sections 4 and 5).

In Figure 3, we have shown the OBDD size trends for the two encodings when the tag size changes (others parameters remaining constant). The space savings with the CAM encoding are similar to that in Figure 3. Although these results are not as dramatic as those of Figure 2, they show that use of the CAM encoding still results in at least an order of magnitude space savings, as compared to the plain encoding.

We can explain the trends in these results in terms of circuit structure, and the interactions of the circuit Boolean functions. Consider the 3-entry CAM described in Section 3.1, and let the tag size be k. In this design the ith match line, $match[i]$ contains the result of a match between the tag input and the ith tag entry. When the plain encoding is used, the ith match line contains the result of the match between the input tag $\vec{tagin}$ and the ith tag entry $\vec{tag}_i$. After the compare, the Boolean function associated with $match[i]$ is $f_{match[i]} = \neg((tagin[k-1] \oplus tag_i[k-1]) \vee \ldots \vee (tagin[0] \oplus tag_i[0]))$ The value on each dataout line, $Dataout[j]$, is a function of all the functions on all the match lines, bit $d[j]$ (used in the associative read assertion), and $\vec{i}$. So, potentially there are interactions among all the Boolean variables associated with the tag and data entries and the tag input.

When the CAM encoding is used, the antecedent fragment (Section 3.1) specifying the 0th tag entry is given by $(when(\vec{i} = 0)(T[0] = \vec{tin})) \wedge (when(\vec{i} \neq 0)(\exists p_1.T[0][p_1] = \neg tin_{p_1}))$. When the tag input is tin, then the 0th tag entry matches only if $\vec{i} = i_1 i_0 = 0$. This is the information conveyed by the Boolean function on $match[0]$. Therefore, $f_{match[0]} = \overline{i_1} \cdot \overline{i_0}$. So, the functions on the dataout lines depend only on the Boolean variables in $\vec{d}$, and $\vec{i}$. Thus, the use of CAM encoding minimizes the variable interaction and this results in substantial space savings, especially when the number of entries is large. We have not shown the running times of the assertions here, most of which finish in a few seconds on a RS/6000™ model 250 workstation.

4. PowerPC Branch Target Address Cache Array

The Branch Target Address Cache (BTAC) array is part of the speculative instruction fetch mechanism on some PowerPC processors. The particular BTAC we verified, from a recent PowerPC processor, was a 64 entry content addressable memory, where each entry consists of a 30-bit tag and a 32-bit data part (Figure 4). The branch address is used to access the BTAC array, which contains the target address of previously executed branch instructions that are predicted to be taken.

The primary task of this unit is an associative read operation, i.e., given a branch instruction address presented at the *rd0_fadr* input, to determine if there is a matching tag entry, and if so give out the corresponding data entry, which is the branch target address. The verification of this operation is similar to that of the CAM associative read operation of Section 3. There are also a number of other operations this unit performs, including reset, and initialization of its round-robin register. Our discussion, however, will focus on the *replace*, or *CAM write* operation.

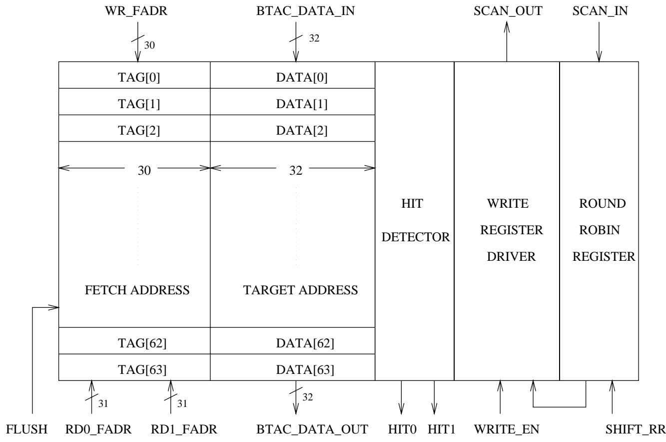

Figure 4: Branch Target Address Cache unit.

4.1. BTAC Replace operation

In the replace operation, a TAG-DATA pair is updated with new values. The selection of an entry for updating is not necessarily based on the address of the entry, rather, it can also be based on a round-robin replacement policy. This operation is essentially a CAM write operation.

The first step in this operation is to select the entry to be replaced. An associative read is done on a tag value presented at read port 1, i.e., *rd1_fadr*. This input tag is compared to all the stored tags in parallel, and if there is a match, *hit1* rises and the matching entry is updated with the new values at *wr_fadr* (the new tag) and *btac_data_in* (the new data). If there is no match, then a round-robin replacement policy is enforced. This replacement policy is implemented with a 64-bit round-robin register (right side of Figure 4) which is a one-hot encoded ring counter. The bit position in the ring counter which is 1 points to the BTAC entry to be replaced in the case of a miss on the address at *rd1_fadr*. Irrespective of the value on *hit1*, all entries which are not replaced remain unchanged.

Verification of the replace operation, required verifying a number of different cases, many of these similar to the memory write operation. One of the more interesting cases is that outlined above, when there is no hit on *rd1_fadr*, and the TAG-DATA entry pointed to by the round-robin register is written to (and all other TAG-DATA entries are unchanged). This case is discussed below.

To verify this case, we encoded the TAG value to be unequal to the symbolic value *tag*. In order to do this, we could use a CAM encoding where the ith bit position of a TAG[] entry equals $\overline{tag[i]}$, and all other bit positions of the TAG[] entry are X. The problem with this is that if we have to show that TAG[] remains unchanged, then it is not sufficient to show that it still has its earlier value which is of the form <X,X,X,...,$\overline{tag[i]}$,...X>. The bit positions which are X can change, and we would not be able to detect it, since X denotes an absence of information. Therefore, in the assertion below, we have a vector of symbolic values called *val*, which we use to encode a value of the form <val[0],val[1],...,val[i-1],$\overline{tag[i]}$,val[i+1],...,val[n-1]>. This value is unequal to *tag*; but, we can also detect whether the value of TAG[] remains unchanged in an operation, since none of the bit positions contain X. In this manner, we verified that only the tag entry pointed to by the round-robin register was updated, and the rest remained unchanged.

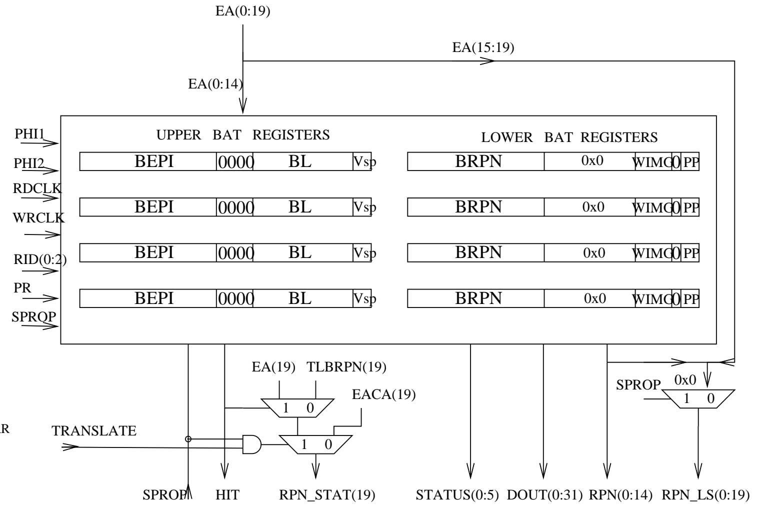

Figure 5: DBAT organization

4.2. Results

The most complex BTAC assertion takes 40MB of memory and 5 minutes to run, on a RS/6000 model 350 workstation. Of this 40MB, 24 MB is taken up by the OBDDs, and the remaining space is taken up by other run-time data structures. The total run time for all assertions was 20 minutes. All the BTAC assertions passed, and no bugs were uncovered in this circuit. If a more naive Boolean encoding had been used for the BTAC verification, the OBDD growth trends of Figures 2 and 3 predict that a memory of several GB, and a 32-bit address space, would not have been sufficient for this verification!

5. PowerPC Block Address Translation array

The PowerPC architecture includes a block address translation (BAT) mechanism which maps ranges of effective addresses larger than a single page into contiguous areas of physical memory [5]. Such areas are used for data not subject to normal virtual memory handling. such as a memory-mapped display buffer. This translation mechanism is implemented as an array consisting of software controlled registers.

The DBAT array implements the BAT translation mechanism for data memory references. It is a CAM containing 4 tag entries and 4 data entries. Each tag-data entry pair is organized as a pair of registers called the Upper DBAT Register and the Lower DBAT register (Figure 5). The two operations this array performs are the SPR ("special purpose register") operation, and the non-SPR operation. In the SPR operation, this array behaves like a register file where in a single clock cycle reads and writes are done on the Special Purpose Registers (SPRs) constituting the upper and lower DBAT registers.

In the non-SPR mode of operation, the DBAT array behaves like a CAM and it translates the 9 to 15 most significant bits of the logical address (bit 0 is the MSB) into the physical address. The remaining bits pass unchanged. In Figure 5, the incoming logical address (top 15 bits, i.e. EA(0:14)) is compared to the block effective page index (BEPI) entry. The block length field (BL) contains a 11-bit mask, used to determine which bits are to be compared. If the mask is all 0's, then all 15 bits are compared, and the corresponding 15-bit data entry, the Block Real Page Number (BRPN), is sent out as the upper 15 bits of the physical address. If the mask entry is all 1's, then only the top most 4 bits are compared, and on a match only the top most 4 bits of the

BRPN are put out, as bits 0 to 3 of the physical address. In this case, bits 4 to 14 of the physical address are copied from the logical address. The mask has a unary-style encoding. The 12 possible legal values for the mask for each tag-data entry are 00000000000, 00000000001,..., 01111111111 and 11111111111. The lower 11 bits of the BEPI and BRPN entries should be 0 in positions where the mask value is 1. Every register pair has a valid bit, Vsp. This bit, when 0, indicates that the BEPI-BRPN-BL entry is invalid, and there can be no match on this entry. The system invariant specified in the PowerPC programming environment manual [5] is that at most one DBAT entry should match the incoming logical address. More details on this complex unit can be found in [5]. While we have verified all the DBAT operations, here we describe only the verification of some aspects of the interesting "non-SPR" mode of operation.

5.1. DBAT non-SPR operation

In section 3.1 we described a way of encoding that a register $T[0]$ was not equal to a value $\vec{tin}$. We abbreviated this by $\exists p.T[0][p] = \neg tin_p$. This encoding does not work directly for expressing a mismatch on an upper DBAT register because comparison can be disabled on some selected register bits by the mask field. Furthermore, the bits masked out can be different for all four of the upper DBAT registers. In order to express that a register contains a data value that does *not* match the incoming data, we needed to take into account the (12) legal values the mask bits can hold.

Using 4 symbolic Boolean variables, $\vec{m} = m_3m_2m_1m_0$, we created a symbolic vector, $\vec{M}$, to encode the 12 legal mask values. Given a vector of symbolic Boolean variables, $\vec{u}[0-14]$, all legal BEPI entries may be expressed symbolically as $\vec{u}[0-3] \; || \; \vec{u}[4-14]\&\neg\vec{M}$, where $||$ is the bitvector concatenation operator. The position of 1's in a mask indicates the BEPI bit positions which are not compared to an incoming tag, $\vec{tin}$. Therefore, if the mask is 00000000111, the comparison is done over bits 0 through 11, and the mismatch can be over any of these 12 bit positions. This mismatch is expressed as $\exists p.(0 \leq p \leq 11) \wedge T[0][p] = \neg tin_p$ Combining such information for all the 12 legal mask values covers all possible cases of a tag mismatch. Since every register pair can have a different mask, we need a separate set of Boolean variables, $\vec{m}$, for encoding the mask value for each pair. Also, for each register pair we need a distinct Boolean variable, v, to indicate whether this entry is valid. Using this encoding, verification of the associative read can be done in a manner similar to that described earlier.

5.2. Results

We wrote two assertions for verifying the DBAT circuit, one for SPR operation, and the other for non-SPR operation. On a RS/6000 model 350 workstation, peak memory requirements for running all the assertions was 16.1 MB, and the total time was 15 minutes. We also wrote an assertion for the non-SPR operation using the plain encoding, to compare against these results. This encoding did not work well. Even with many control signals set to non-symbolic values, the memory required was over 100 MB!

We discovered two bugs in this circuit, both in the SPR mode of operation. The first bug was that the signal, rpn_ls, should have been all 0's, and was not. The second bug involved an incorrect implementation of the signal, rpn_status19. It is significant that these bugs were discovered by running just one assertion specifying the SPR operation. This is in contrast to the commonly accepted practice of running a huge number of (non-symbolic) simulation vectors, often for days, with no certainty that such corner cases will be brought out.

6. Conclusion

We have reported on new techniques to verify CAMs. It is based on symbolic trajectory evaluation and new Boolean encoding techniques. We have shown that our techniques avoid the OBDD space explosion problem for CAMs, and the OBDD space scales linearly or sub-linearly with increasing in various CAM parameters. Using these techniques we have verified complex CAMs from a recent PowerPC microprocessor. This work opens the way to the efficient verification of numerous on-chip CAMs such as TLBs, cache tags and branch target buffers.

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] R. E. Bryant, "Formal Verification of Memory Circuits by Switch-Level Simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.10, no.1, Jan. 1991; pp. 94-102.

[3] N. Ganguly, M. Abadir, M. Pandey "PowerPC Array Verification Methodology using Formal Techniques," in *Proceedings of the International Test Conference*, 1996.

[4] K. L. McMillan. "Symbolic model checking - an approach to the state explosion problem," PhD thesis, SCS, Carnegie Mellon University, 1992.

[5] "PowerPC™ Microprocessor Family: The Programming Environments," Motorola Inc., 1994.

[6] M. Pandey, R. Raimi, D. Beatty, R. Bryant, "Formal verification of PowerPC(TM) arrays using symbolic trajectory evaluation," *Proc. 33rd ACM/IEEE DAC*, 1996.

[7] 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).

[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, 1993.

[9] N. Weste, K. Eshraghian, *Principles of CMOS VLSI design, A systems Perspective*, Second Edition, Addison Wesley, 1994.

PowerPC and RS/6000 are trademarks of the International Business Machines Corporation used under license therefrom.