

Automatic Generation of Assertions for Formal Verification of PowerPC™ * Microprocessor Arrays Using Symbolic Trajectory Evaluation

Li-C. Wang, Magdy S. Abadir, and Nari Krishnamurthy
Somerset PowerPC Design Center
Motorola, Inc., Austin, Texas

Abstract

For verifying complex sequential blocks such as microprocessor embedded arrays, the formal method of symbolic trajectory evaluation (STE) has achieved great success in the past [[3], [5], [6]]. Past STE methodology for arrays requires manual creation of “assertions” to which both the RTL view and the actual design should be equivalent. In this paper, we describe a novel method to automate the assertion creation process which improves the efficiency and the quality of array verification. Encouraging results on recent PowerPC arrays will be presented.

1 Introduction

Arrays (such as cache, cache tag, tables, register files, etc.) are complex sequential logic blocks which represent one of the major challenges in modern microprocessor design environment. The challenge is twofold: the size of these arrays and the complexity of their designs. In today’s RISC microprocessors, arrays may contain up to 80% of the total transistors in use. Verifying their correctness is obviously very important. Usually, the RTL view of an array design is first simulated as part of the full-chip verification effort. Then, the transistor level schematic view is verified to be equivalent to the corresponding RTL view during logic verification. In the past, logic verification relied on vector simulation and boolean equivalence checking. While exhaustive vector simulation is not feasible for arrays due to the exponentially large number of patterns required, boolean equivalence checking is not suitable either for it fails to capture the complex timing and sequential control [4].

In recent years, at the PowerPC Design Center, Somerset, an additional formal verification methodology has been deployed for arrays. The methodology utilizes the formal verification technique Symbolic Trajectory Evaluation (STE) [7] to verify the equivalence between the RTL view and the schematic view [[3], [5], [6]].

*IBM and PowerPC are trademarks of IBM Corporation in the United States, or other countries, or both

In STE verification, a set of so-called *assertions* are **manually** created according to the array specification, usually in the RTL view. Then, both the RTL view and the transistor level schematic view are proved to satisfy the same set of assertions under STE. Creating assertions usually requires a deep understanding of the overall functionality of the array so that each array operation can be expressed properly into an assertion. Such a process can be very tedious and potentially incomplete. In this paper, we will describe a novel framework for automatic generation of assertions directly from the RTL view for arrays and hence, provide a more efficient and robust approach for logic verification of arrays. Encouraging results on five selected array designs in recent PowerPC microprocessors will be reported.

2 Background

Arrays comprise storage elements, together with complex timing and control logic that determine how they are used and updated. An array design can be represented by different views, each for a particular set of tools and applications. For example, the RTL view is the most abstract, which serves as the high-level specification of a design and is mainly used for functional simulation. The actual implementation is represented by the transistor schematic view.

2.1 Logic Verification with STE

The main purpose of logic verification is to verify the equivalence between the RTL view and the schematic view. At Somerset, this is achieved by vector simulation, boolean equivalence checking and STE verification. For arrays, STE verification is used.

STE [7] is a formal verification technique which is a descendant of symbolic simulation [1]. At the core of symbolic simulation, Ordered Binary Decision Diagram [2] is used to efficiently manipulate boolean functions. In STE, a circuit is specified in terms of a set of properties in a restricted temporal logic form. Properties are expressed by so-called **assertions** which are of the form “**Antecedent** \implies **Consequent**” where both “Antecedent” and “Consequent” consist of a number of formulae. Formulae can be simple predicates such as “line A is ‘a’” (line A holds the symbolic value ‘a’ at current time), or conjunctions of these simple predicates. Formulae can be applied with the next time operator in order to state the facts such as “line A remains ‘a’ in the next time step.” It is also possible to apply the *domain* operator so that “line A remains ‘a’ **when D**

is true” can be expressed. This simple logic in STE is sufficient for array verification purposes. In STE, conditions specified in the antecedents are symbolically simulated, and the consequents are asserted. (Please refer to the previous papers [[3] [5] [6]] for details)

2.2 Assertion Creation

For complex array designs, assertion creation can be a complicated, tedious, and unreliable process. First, a person needs to fully understand the functionality of a unit before creating the assertions. Assertions themselves can be long and tedious in order to specify all aspects of a design in details. Very often, a verification engineer would create assertions to verify only what he/she understood as the normal operations. Hence, assertions can be incomplete. To avoid all these problems, a way to automatically generate assertions is required.

3 Automatic Assertion Generation for Arrays

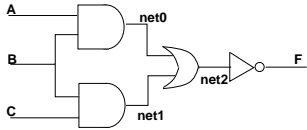


Figure 1: A simple combinational circuit

Automatic assertion generation for sequential designs in general is a difficult problem. Such a tool must be able to parse the statements in the RTL view, automatically extract the high level properties of the design, and then express them in the antecedent/consequent form. On the other hand, generating assertion for a combinational design is rather straightforward. Consider the combinational logic shown in Figure 1. The assertion for this circuit can be stated as the following.

Antecedent:	\implies	Consequent:
$A = a, B = b,$	(lead to)	let $net0 = A \& B$
$C = c$		let $net1 = B \& C$
		let $net2 = net0 \mid net1$
		$F = \neg net2$

where $net0, net1, net2$ are temporary variables which hold the temporary symbolic expressions for the final consequent, and a, b, c are all symbolic values. Based on the simple observation above, the idea of our automatic assertion generation approach for arrays is to *abstract out the timing and sequential elements so that they can be treated as a combinational circuit at a given time*. At Somerset, a tool called AGA (automatic generation of assertion) is being developed for arrays.

3.1 Define Timing

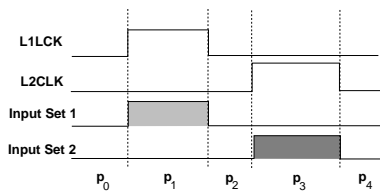


Figure 2: An example of clocks

The only necessary user input to AGA is a clock definition where the resulting assertion should be based on. For example, a typical array may operate based on two non-overlapping clocks, L1CLK and L2CLK. A clock definition is illustrated in Figure 2. The two clocks divide the cycle into five different phases of interest. Since arrays are clock-based, they behave like a combinational circuit within each phase. Without additional information, AGA will automatically assign different symbolic values to all primary inputs at all different phases, and then try to derive consequents for each phase *independently* based on the RTL view. Most of the time, this is unnecessary and quite inefficient. Therefore, as the Figure shows, we allow user to specify optional information regarding when to supply “meaningful” symbols to what inputs. Note that any unspecified inputs at a given phase will be assumed to have the unknown value X. An X at an input really means “don’t care.” Note that a different clock definition can be defined in the Figure by interchanging L1CLK and L2CLK.

3.2 Use Temporary Variables for Each Clock Phase

Basically, AGA behaves like a translator that translates each RTL construct into a sequence of intermediate symbolic expression assignments (as those in the combinational logic example). The translation process follows the topological order of a design from primary inputs to outputs. Final consequents will be formed at the primary outputs and at the memory elements as well.

For each variable name used in the RTL view, which can represent an input, an output, an internal line, or a memory block, a list of temporary variables will be created in AGA. For a variable A , these temporary variables will be involved in a list of symbolic expression assignments and will all be of the form “ $A_{[p_i, cond]} = exp$ ” which means that A at the end of phase p_i will hold a symbolic value given by the boolean expression exp **when** $cond$ is true. As a simple example, supposed primary input Din is given a symbolic value $Data$ during $p2$. This results in five temporary variables inside AGA (one for each clock phase): $Din_{[p0,1]} = Din_{[p1,1]} = Din_{[p3,1]} = Din_{[p4,1]} = X$ and $Din_{[p2,1]} = Data$.

3.3 Combinational Logic

Except for latch and memory element, other constructs in the RTL view can be treated as part of the combinational logic. These include assignment statement, if-then-else statement, case statement, tri-state buffer, etc. Note that Somerset’s RTL design language does allow the for-loop statement. However, for-loop is considered a statement for macro expansion, which is commonly used to describe a piece of replicated logic. In the following, we will consider a few examples to illustrate the fundamental ideas. Note that the translation process handles combinational statements for every clock phase independently.

Expression involving X

Assume that $B_{[p_i,1]} = X$. The following rules tell how to deal with unknown values X during the translation process.

RTL statements	AGA results at p_i
$A = exp \& B$	$\longrightarrow A_{[p_i, \neg exp]} = 0$
$A = exp \mid B$	$\longrightarrow A_{[p_i, exp]} = 1$
$A = \neg B$	$\longrightarrow A_{[p_i, 1]} = X$
$A = B$	$\longrightarrow A_{[p_i, 1]} = X$

When conditions

Suppose that $A_{[p_i,C_1]} = \text{exp1}$ and $B_{[p_i,C_2]} = \text{exp2}$. The following rules tell how to merge the “when” conditions through logical expressions.

RTL	AGA results at p_i
$C = A \& B$	$\rightarrow C_{[p_i,(\neg \text{exp1} \& C_1 \& \neg C_2) (\neg \text{exp2} \& C_2 \& \neg C_1)]} = 0$ $C_{[p_i,C_1 \& C_2]} = \text{exp1} \& \text{exp2}$
$C = A B$	$\rightarrow C_{[p_i,(\text{exp1} \& C_1 \& \neg C_2) (\text{exp2} \& C_2 \& \neg C_1)]} = 1$ $C_{[p_i,C_1 \& C_2]} = \text{exp1} \text{exp2}$
$C = \neg A$	$\rightarrow C_{[p_i,C_1]} = \neg \text{exp1}$
$C = A$	$\rightarrow C_{[p_i,C_1]} = \text{exp1}$

Note that all variables use the same phase index p_i . Since any boolean expression assignment can break down into a sequence of simple assignments involving binary AND and OR, and unary NOT, the above table is sufficient for more general cases.

If-Then-Else

Suppose that exp below has been evaluated first and resulted in a variable $E_{[p_i,C_0]} = \text{exp0}$, and $A1_{[p_i,C_1]} = \text{exp1}$, $A2_{[p_i,C_2]} = \text{exp2}$. The following presents the idea of how to handle the if-then-else statement.

RTL	AGA results at p_i
if (exp) then	$\rightarrow A_{[p_i,C_0 \& C_1 \& \neg C_2 \& \text{exp0}]} = \text{exp1}$
$A = A1$	$A_{[p_i,C_0 \& C_2 \& \neg C_1 \& \neg \text{exp0}]} = \text{exp2}$
else $A = A2$	$A_{[p_i,C_0 \& C_1 \& C_2]} = (\text{exp0} \& \text{exp1}) (\neg \text{exp0} \& \text{exp2})$

Tri-state buffer and bus

Let “(exp) \leftarrow (select)” denote a tri-state buffer with data input exp and switch line select . Also, let $A1_{[p_i,C_1]} = \text{exp1}$, $A2_{[p_i,C_2]} = \text{exp2}$. To simplify our presentation, we assume two sources with select lines $S1_{[p_i,1]} = s1$, $S2_{[p_i,1]} = s2$, respectively. In this example, we assume also that the bus is a wired-OR device.

RTL statements	AGA results at phase p_i
$\text{Bus} = (A1) \leftarrow S1$	$\rightarrow \text{Bus}_{[p_i,s1 \& \neg s2 \& C_1 \& \neg C_2]} = \text{exp1}$ $\text{Bus}_{[p_i,s2 \& \neg s1 \& C_2 \& \neg C_1]} = \text{exp2}$
$\text{Bus} = (A2) \leftarrow S2$	$\text{Bus}_{[p_i,s1 \& s2 \& C_1 \& \neg C_2 \& \text{exp1}]} = 1$ $\text{Bus}_{[p_i,s1 \& s2 \& C_2 \& \neg C_1 \& \text{exp2}]} = 1$ $\text{Bus}_{[p_i,(s1 \& s2) \& C_1 \& C_2]} = s1 \& \text{exp1} s2 \& \text{exp2}$

Summary

In summary, processing a statement of combinational logic is straightforward if no “when” condition is involved. For a particular line (or variable) in the RTL view, a “when” condition defines an unique temporary variable in AGA at a given phase. In general, when a single statement involves multiple variables, each with a list of temporary variables, all possible combinations of their “when” conditions should be exhausted. In this case, all new “when” conditions produced should be mutually exclusive (see the Bus example).

3.4 Latch

The translation for latch is similar to that for the tri-state buffer. Normally, latches are controlled by the input clocks. Take the clock setting in Figure 2. A latch can be easily processed as the example shown below (Assume $A_{[p_i,C_i]} = \text{exp}$ has been obtained for A at phase p_i).

RTL statements	AGA results
$D = \text{latch}(A, L2CLK)$	$\rightarrow D_{[p_0,1]} = D_{[p_1,1]}$ $= D_{[p_2,1]} = X$ $D_{[p_3,C_3]} = \text{exp}$ $D_{[p_4,C_3]} = D_{[p_3,C_3]} = \text{exp}$

In other words, a latch will hold its state unless its input clock becomes active. Initially, its state is the unknown X. This example also illustrates how the temporary variables can be used to carry unchanged latch states across the clock phases ($D_{[p_4,C_3]} = D_{[p_3,C_3]}$).

3.5 Memory Element

The most difficult part in the translation process is to handle memory elements. However, this problem can be simplified by imposing some useful design rules on arrays. At Somerset, arrays in the RTL view are usually designed with pre-defined RAM primitives. In this way, all memory elements can be grouped into different RAM blocks, each implemented by a RAM primitive where its functions are well defined and understood. In AGA, a set of translation rules are associated with a RAM primitive. To illustrate the idea, we show the rules for a simple RAM primitive below. Figure 3 shows the RAM model, called as P-RAM (primitive RAM). Due to space limitations, other more complex primitives are omitted.

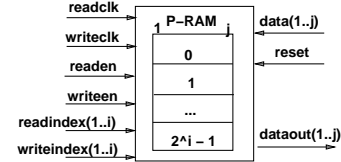


Figure 3: RAM primitives

The assumption for P-RAM is that read and write cannot happen simultaneously. However, this does not exclude us from verifying both read and write operations in one pass of the symbolic simulation. Usually, readclk and writeclk are derived from the input clocks and in a normal design, both can only be active during a particular clock phase. Note that the user can define how the primitive behaves when it is not doing a read. To simplify our presentation, we assume that dataout is precharged to 1 while the P-RAM is not doing a read.

Again, the goal of AGA translation is to find a sequence of symbolic expression assignments to model the behavior of P-RAM. To do so, we need to assume that the P-RAM contains some initial symbolic values. In STE, this can be achieved with symbolic index by declaring that

$$\text{P-RAM}[\text{Addr}(1..i)] = D(1..j)$$

Usually, this initialization is assumed to be done at the beginning of phase p_0 .

Similarly, the translation process handles the P-RAM independently for each clock phase. For simplicity, assume that $\text{writeclk}_{[p_3,1]} = \text{wexp}$, $\text{readclk}_{[p_1,1]} = \text{rexp}$ (possible write in p_3 and read in p_1), $\text{readen}_{[p_1,1]} = \text{ren}$, $\text{writen}_{[p_3,1]} = \text{wen}$, $\text{writeindex}_{[p_3,1]} = \text{windex}$, $\text{readindex}_{[p_1,1]} = \text{rindex}$, $\text{data}_{[p_3,1]} = \text{da}$, and $\text{reset}_{[p_i,1]} = \text{re}_i$ for $i = 0, 1, 2, 3, 4$. Note that it is possible to include a “when” condition in each of the above variables. Then, all possible combinations of these conditions will be exhausted and more temporary variables will be produced.

In order to properly specify the memory content, the temporary variables created for P-RAM will include a third entry in its footnote index, which is the index Addr used for initialization. Therefore, $\text{P-RAM}_{[p_i,C,Addr]} = D$ can be read as “P-RAM[Addr] = D during p_i when C is true.”

$$\begin{array}{l}
R_i = re_0 \mid \dots \mid re_i \text{ for all } i \\
\hline
dataout_{[p_1, (rindex=Addr) \& rexp \& ren \& \neg R_1]} = D \\
\quad \quad \quad dataout_{[p_1, R_1]} = 0 \\
\quad \quad \quad dataout_{[p_j, 1]} (j = 0, 2, 3, 4) = 1 \\
\hline
P\text{-RAM}_{[p_j, \neg R_j, Addr]} = D \quad (j = 0, 1, 2) \\
P\text{-RAM}_{[p_3, we xp \& wen \& \neg R_3, Addr]} = da(windex = Addr) \\
\quad \quad \quad \mid D(windex \neq Addr) \\
P\text{-RAM}_{[p_4, \neg R_4, Addr]} = P\text{-RAM}_{[p_3, we xp \& wen \& \neg R_3, Addr]} \\
P\text{-RAM}_{[p_j, R_j, Addr]} = 0 \quad (j = 0, 1, 2, 3, 4)
\end{array}$$

As discussed earlier, an array may contain multiple instances of the RAM primitives. One problem is how to initialize these RAM instances properly. AGA decides how to initialize multiple RAM instances by checking how their address inputs and data inputs are related. There are three possible scenarios: 1) Two RAMs share the same data inputs. Usually, this case represents a bad usage of RAM primitive where each RAM instance implements part of the same memory block, and it should be implemented with just one RAM. 2) Two RAMs share the same address inputs. Both RAMs will be initialized using the same symbolic address, but with different symbolic data. 3) No sharing occurs. In this case, each RAM is independent and hence, is initialized with a unique symbolic address and data.

3.6 Forming the Final Consequents

After the AGA translation process is done for all primary outputs, a list of temporary variables have been created for every line (variable) and every memory block (RAM primitive) used in the RTL view. Forming the final consequents is straightforward. For arrays, consequents are collected at the RAM instances and at all primary outputs. For each RAM instance, a temporary variable will be converted into a consequent statement in the STE. For example, “P-RAM_[p₄, R₄, Addr] = 0” will be translated into a consequent statement “P-RAM[Addr] are all 0 **when** R₄ is true before the end of phase p₄.” Note that no consequent will be given to a variable which is assigned with the value unknown X. Moreover, each consequent only checks the corresponding result close to the end of each clock phase so that it is guaranteed all results due to input changes during that phase has stabilized before reaching the phase end.

4 Summary and Results

Five selected arrays, a data segment register (DSR), an instruction cache tag (ITAG), a branch target instruction cache (BTIC), a translation lookaside buffer (TLB), and a block address translation (BAT) unit, are used as our benchmarks for the initial AGA exercise.

Table 1 presents the comparisons of the overall run time (on the schematic view) between manual assertions and automatic assertion (on an IBM RS6000/590 machine with 512M memory). A significant amount of time and efforts were spent to manually create these assertions while the costs of the automatic approach was minimal. Also note that a fixed OBDD variable ordering (which does not favor either automatic or manual assertions) is enforced for each array to make sure a fair comparison in our experiments.

For all cases, automatic assertions outperform manual assertions in the actual verification run time as well. The reason for these results is that manual assertions are usually not mutually exclusive with respect to which part of the circuit they try to verify.

Arrays	Manual	Automatic
DSR	84 secs	72 secs
BAT	7455 secs	6751 secs
TLB	1426 secs	575 secs
BTIC	1546 secs	768 secs
ITAG	5346 secs	3189 secs

Table 1: Run time comparison between manual assertions and automatic assertion for the five arrays

It is interesting to note that for each case, we have observed that some consequents generated for the automatic assertion are not considered by any of the manual assertions, but not vice versa. Hence, in every comparison result in the two Tables, the automatic assertion actually verifies more properties than the manual assertions.

5 Conclusion

STE has been shown to be an effective technique for logic verification of microprocessor embedded arrays. The past STE methodology for array verification requires manual creation of assertions. Since manual creation of assertions can be tedious and incomplete, in this paper we have presented a framework to automatically generate assertions for arrays from the RTL view. Our experience indicates that the new approach is very effective and can provide more complete verification results.

References

- [1] Randal E. Bryant, *Symbolic simulation — techniques and applications*, 27th Design Automation Conference, 1990.
- [2] Randal E. Bryant, *Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams*, ACM Computing Surveys, Vol. 24, No. 3, Sep. 1992.
- [3] Neeta Ganguly, Magdy S. Abadir, and Manish Pandey, *PowerPC Array Verification Methodology Using Formal Verification Techniques*, International Test Conference, Washington DC., 1996. pp. 857-864.
- [4] Charles H. Malley and M. Dieudonne, *Logic Verification Methodology for PowerPC Microprocessors*, 32nd Design Automation Conference, 1995, pp. 234-240.
- [5] Manish Pandey, Richard Raimi, Derek L. Beatty, and Randal E. Bryant, *Formal verification of PowerPCTM arrays using symbolic trajectory evaluation*, Proc. 33rd Design Automation Conference, Las Vegas, NV., 1996.
- [6] Manish Pandey, Richard Raimi, Randal E. Bryant, and Magdy S. Abadir *Formal verification of Content Addressable Memories Using Symbolic Trajectory Evaluation*, Proc. 34rd Design Automation Conference, 1997.
- [7] C.J.H. Seger and Randal E. Bryant, *Formal verification by symbolic evaluation of partially-ordered trajectories*, Formal Methods in System Design 6, 1995, pp. 147-189.