Appears in the proceedings of *Computer-Aided Verification '97*, O. Grumberg, ed., LNCS 1254, Springer-Verlag, June 1997, pp. 388-399.

Efficient Modeling of MemoryArrays in Symbolic Simulation[1]

Miroslav N. Velev
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
mvelev@ece.cmu.edu

Randal E. Bryant
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
randy.bryant@cs.cmu.edu

Alok Jain
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
alok.jain@ece.cmu.edu

Abstract. This paper enables symbolic simulation of systems with large embedded memories. Each memory array is replaced with a behavioral model, where the number of symbolic variables used to characterize the initial state of the memory is proportional to the number of memory accesses. The memory state is represented by a list containing entries of the form $\langle c, a, d \rangle$, where c is a Boolean expression denoting the set of conditions for which the entry is defined, a is an address expression denoting a memory location, and d is a data expression denoting the contents of this location. Address and data expressions are represented as vectors of Boolean expressions. The list interacts with the rest of the circuit by means of a software interface developed as part of the symbolic simulation engine. The interface monitors the control lines of the memory array and translates read and write conditions into accesses to the list. This memory model was also incorporated into the Symbolic Trajectory Evaluation technique for formal verification. Experimental results show that the new model significantly outperforms the transistor level memory model when verifying a simple pipelined data path.

1. Introduction

Simulation is widely used to validate systems at various levels of design abstraction such as the transistor, gate, and behavioral levels. The normal simulation models for memory arrays at all these levels explicitly represent each memory bit. This is not a problem for conventional simulation which uses a single logic value to denote the state of a memory bit. However, symbolic simulation would require a symbolic variable for every bit of the memory. In addition, bit-level symbolic model checking would need the next-state function for each memory bit. Therefore, the number of variables in symbolic computation is proportional to the size of the memory, and is prohibitive for large memory arrays.

This paper shows a way to overcome this limitation by replacing each memory array with an Efficient Memory Model (EMM). The EMM is a behavioral model, which allows the number of symbolic variables used to be proportional to the number of memory accesses rather than to the size of the memory. It is based on the observation that a single execution sequence typically contains a limited number of memory accesses.

Symbolic Trajectory Evaluation (STE) is an extension of symbolic simulation that has been used to formally verify circuits [8]. STE has been applied on the verification of a simple pipelined data path [2]. Incorporation of the EMM in STE enabled us

1. This research was supported in part by the SRC under contract 96-DC-068.

to verify the pipelined data path with a significantly larger register file than previously possible.

Symbolic model checking has also been used to verify a pipelined data path [3]. However, the limitation of the method is that it requires the next-state relation for the entire circuit, which leads to introducing two symbolic variables for every state bit in the circuit. Burch, Clarke, and Long [4] represent the transition relation as an implicit conjunction of transition relations for parts of the circuit. In this way, they avoid building a monolithic BDD for the transition relation of the entire circuit, but still need two symbolic variables for each memory bit. Clarke, Grumberg, and Long [6] propose a method for using abstraction in order to reduce the complexity of symbolic model checking. They show how abstraction functions can be applied to produce an abstract model whose state space is a subset of that of the original model, such that if certain properties hold on the abstract model, they will also be true for the original one. However, this requires a careful choice of abstraction functions by the user.

A symbolic representation of memory arrays has been used by Burch and Dill [5]. Their technique is also based on symbolic simulation. However, it verifies only the control, assuming that the combinational logic in the data path is correct. On the other hand, our method verifies the entire circuit. While Burch and Dill use uninterpreted functions with equality, which abstract away the details of the data path, we use BDDs and model fully the entire circuit, but that leads to greater memory and CPU time consumption. The logic of uninterpreted functions with equality allows them to introduce only a single symbolic variable for denoting the initial state of a memory array. The need to have data at the bit level in order to verify the data path, requires the user of our method to introduce symbolic variables proportional to the number of memory array accesses. Given a real circuit, their method would require the user to provide the distinction between the control and the data path. Ours would need only an identification of the memory arrays. Finally, we perform the verification at the circuit level of the implementation, while they operate on an abstracted high level model of the control and require the availability of an appropriate compiler to automatically extract the control from the real circuit.

This paper advocates a two step approach for the verification of circuits with large embedded memories. The first step is to use STE to verify the transistor level memory arrays independently from the rest of the circuit. Pandey and Bryant have combined symmetry reductions and STE to enable the verification of very large memory arrays at the transistor level [7]. The second step is to use STE to verify the circuit after the memory arrays are replaced by EMMs.

Section 2 describes the symbolic domain used in our algorithms. Section 3 gives a brief overview of STE. Sections 4 presents the EMM and section 5 introduces its underlying algorithms. Section 6 explains the way to incorporate the EMM into STE. Experimental results are presented in Section 7, and plans for future work are outlined in Section 8.

2. Symbolic Domain

We will consider three different domains - Boolean, address, and data - corresponding respectively to the control, address, and data information that can be applied at the inputs of a memory array. Symbolic variables will be introduced in each of the domains and will be used in expression generation. Address and data expressions will be represented by vectors of Boolean expressions having width n and w, respectively, for a memory with $N = 2^n$ locations, each holding a word consisting of w bits. The types **BExpr**, **AExpr**, and **DExpr** will denote respectively Boolean, address, and data expressions in the algorithms to be presented.

We will use the term *context* to refer to an assignment of values to the symbolic variables. A Boolean expression can be viewed as defining a set of contexts, namely those for which the expression evaluates to **true**.

The selection operator *ITE* (for "If-Then-Else"), when applied on three Boolean expressions, is defined as

$$ITE(b, t, e) \doteq (b \wedge t) \vee (\neg b \wedge e) \tag{1}$$

Address comparison is then implemented as:

$$A1 = A2 \doteq \neg \bigvee_{i=1}^{n} A1_i \oplus A2_i \tag{2}$$

while address selection $A1 \leftarrow ITE(b, A2, A3)$ is implemented by selecting the corresponding bits:

$$A1_i \leftarrow ITE_i(b, A2, A3) \doteq A1_i \leftarrow (b \wedge A2_i) \vee (\neg b \wedge A3_i), \quad i = 1, \ldots, n \tag{3}$$

The definition of data operations is similar, but over vectors of width w.

Although we have used BDDs to represent the Boolean expressions in our implementation, there is nothing about this work that intrinsically requires it to be BDD based. Any canonical representation of Boolean expressions can be substituted.

3. STE Background

STE is a formal verification technique based on symbolic simulation. For the purpose of this paper, it would suffice to say that STE is capable of verifying circuit properties, described as *assertions*, of the form $A \overset{\text{LEADSTO}}{\Rightarrow} C$. The *antecedent* A specifies constraints on the inputs and the internal state of the circuit, and the *consequent* C specifies the set of expected outputs and state transitions. Both A and C are formulas that can be defined recursively as:

1) *a simple predicate*: the three possibilities being (**node** *n* = *boolean_expression*), or (**node_vector** *N* = *address_expression*), or (**node_vector** *N* = *data_expression*), where in the last two cases each node of the node vector *N* gets associated with its corresponding bit-level Boolean expression of the given address or data expression;

2) *a conjunction of two formulas*: $F_1 \wedge F_2$ is a formula if F_1 and F_2 are formulas;

3) *a domain restriction*: (*boolean_expression* $\rightarrow F$) is a formula if F is a formula, meaning that F should hold for the contexts in which *boolean_expression* is **true**;

4) *a next time operator*: $\mathbf{N}F$ is a formula if F is a formula, meaning that F should hold in the next time period.

A shorthand notation for k nested next time operators is $\mathbf{N}^k$. A formula is said to be *instantaneous* if it does not contain any next time operators. Any formula F can be rewritten into the form $F_0 \wedge \mathbf{N}F_1 \wedge \mathbf{N}^2F_2 \wedge \ldots \wedge \mathbf{N}^kF_k$, where each formula F_i is instantaneous. For simplicity in the current presentation, we will assume that the antecedent is free of self inconsistencies, i.e. it cannot have a node asserted to two complementary logic values simultaneously.

STE maintain two global Boolean expressions OK_A and OK_C, which are initialized to be **true**. The STE algorithm updates the circuit node values and the global Boolean expressions at every simulation time step. The antecedent defines the stimuli and the consequent defines the set of acceptable responses for the circuit. The expression OK_A maintains the condition under which the circuit node values are compatible with the values specified by the antecedent. The expression OK_C maintains the condition under which the circuit node values belong to the set of acceptable values specified by the consequent. The Boolean expression $\neg OK_A \vee OK_C$ defines the condition under which the assertion holds for the circuit.

4. Efficient Modeling of Memory Arrays

The main assumption of our approach is that every memory array can be represented, possibly after the introduction of some extra logic, as a memory with only write and read ports, all of which have the same numbers of address and data bits, as shown in Figure 1.

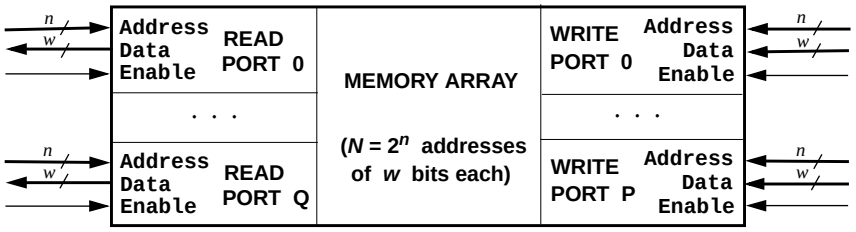


Figure 1. View of a memory array, according to our model.

The interaction of the memory array with the rest of the circuit is assumed to take place on the rising edge of a port `Enable` signal. In case of multiple port `Enable`s having rising edges simultaneously, the resulting accesses to the memory array will be ordered according to the priority of the ports.

During symbolic simulation, the memory state is represented by a list containing entries of the form $\langle c, a, d \rangle$, where c is a Boolean expression denoting the set of con-

texts for which the entry is defined, *a* is an address expression denoting a memory location, and *d* is a data expression denoting the contents of this location. The context information is included for modeling memory systems where the *Write* and *Read* operations may be performed conditionally depending on the value of a control signal. Initially the list is empty.

The list interacts with the rest of the circuit by means of a software interface developed as part of the symbolic simulation engine. The interface monitors the port `Enable` lines. Should a rising edge occur at a port `Enable`, a *Write* or a *Read* operation will result, as determined by the type of the port. The Boolean expression *c* for the contexts of the memory operation will be formed as the condition for a rising edge on the port `Enable`. The operation will be performed if *c* is a non-zero Boolean expression. The `Address` and `Data` lines of the port will be scanned in order to obtain the address expression *a* and the data expression *d*, respectively. A *Write* operation completes with the insertion of the entry $\langle c, a, d \rangle$ in the list. A *Read* operation retrieves from the list a data expression *rd* that represents the data contents read from the memory at address *a* given the contexts *c*. The software interface completes the *Read* operation by asserting the `Data` lines of the port to the data expression *ITE*(*c*, *rd*, *d*), i.e. to the retrieved data expression *rd* under the contexts *c* of the operation and to the old data expression *d* otherwise. The routines needed by the software interface for accessing the list are presented next.

5. Implementation of Memory Operations

5.1 Support Operations

The list entries are kept in order from *head* (low priority) to *tail* (high priority). Entries may be inserted at either end, using procedures *InsertHead* and *InsertTail*, and may be deleted using procedure *Delete*. The function *Valid*, when applied to a Boolean expression, returns **true** if the expression is valid, i.e., true for all contexts, and **false** otherwise. Note that in all of the algorithms, a Boolean expression cannot be used as a control decision in the code, since it will have a symbolic representation. On the other hand, we can make control decisions based on whether or not an expression is valid.

The function *GenDataExpr* generates a new data expression, whose variables are used to denote the initial state of memory locations that are read before ever being written.

5.2 Implementation of Memory *Read* and *Write* Operations

The *Write* operation, shown as a procedure in Figure 2, takes as arguments a memory list, a Boolean expression denoting the contexts for which the write should be performed, and address and data expressions denoting the memory location and its desired contents, respectively. As the code shows, it is implemented by simply inserting an element into the *tail* (high priority) end of the list, indicating that this entry should overwrite any other entries for this address. As an optimization, it removes any list elements that for all contexts are overwritten by this operation. Note that this optimization need not be performed, as will become apparent after the definition of the

Read operation. We could safely leave any overwritten element in the list.

```
procedure Write(List mem, BExpr c, AExpr a, DExpr d)
{ Write data d to location a under contexts c }
  { Optional optimization }
  for each ⟨ec, ea, ed⟩ in mem do
    if Valid(ec ⇒ [c ∧ a=ea]) then
        Delete(mem, ⟨ec, ea, ed⟩)
  { Perform Write }
  InsertTail(mem, ⟨c, a, d⟩)
```

Figure 2. Implementation of the *Write* operation.

The *Read* operation is shown in Figure 3 as a function which, given a memory list, a Boolean expression denoting the contexts for which the read should be performed, and an address expression, returns a data expression indicating the contents of this location.

```
function Read(List mem, BExpr c, AExpr a): DExpr
{ Read from location a under contexts c }
  g ← GenDataExpr()
  return ReadWithDefault(mem, c, a, g)

function ReadWithDefault(List mem, BExpr c, AExpr a, DExpr d): DExpr
{ Attempt to read from location a, using d for contexts where no value found }
  rd ← d
  found ← false
  for each ⟨ec, ea, ed⟩ in mem from head to tail do
      match ← ec ∧ a=ea
      rd ← ITE(match, ed, rd)
      found ← found ∨ match
  if ¬Valid(found) then
      InsertHead(mem, ⟨c, a, d⟩)
  return rd
```

Figure 3. Implementation of the *Read* operation.

The main part of the *Read* operation is implemented with the function *ReadWithDefault*, which will also be used in the implementation of two STE procedures, to be presented in Section 6. The purpose of *ReadWithDefault* is to construct a data expression giving the contents of the memory location denoted by its argument address expression. It does this by scanning through the list from lowest to highest priority, adding a selection operator to the expression that chooses between the list element's

data expression and the previously formed data expression, based on the match condition. It also generates a Boolean expression *found* indicating the contexts for which a matching list element has been encountered. *ReadWithDefault* has as its fourth argument a "default" data expression to be used when no matching list element is found. When this case arises, a new list element is inserted into the *head* (low priority) end of the list.

The *Read* operation is implemented by calling *ReadWithDefault* with a newly generated symbolic data expression *g* as the default. The contexts for which *ReadWithDefault* does not find a matching address in the list are those for which the addressed memory location has never been accessed by either a read or a write. The data expression *g* is then returned to indicate that the location may contain arbitrary data. By inserting the entry $\langle c, a, d \rangle$ into the list, we ensure that subsequent reads of this location will return the same expression. Note that computing and testing the validity of *found* is optional. We could safely insert the list element unconditionally, although at an increased memory usage.

6. Incorporation into STE

Efficient modeling of memory arrays in STE requires that formulas of the form $(c \rightarrow (mem[a] = d))$, where *c* is a Boolean expression, *a* is an Address expression, *d* is a Data expression, and *mem* is a memory array, be incorporated into the STE algorithm described in Section 3. When such formulas occur in the antecedent, they should result in asserting the memory state at location *a* to data *d* given contexts *c*, and are processed by procedure *AssertMem*, presented in Figure 6. Similarly, when such formulas occur in the consequent, they should result in checking the memory state at location *a* for having data *d* given contexts *c*, and are processed by procedure *CheckMem*, presented in Figure 7. The latter is a modified version of function *ReadWithDefault*, with the difference being that it does not insert a new entry into the list when the expression *found* is not valid.

procedure *AssertMem*(**List** *mem*, **BExpr** *c*, **AExpr** *a*, **DExpr** *d*)
{ Determine conditions under which location *a* was asserted to data *d* given
contexts *c*, and reflect them on OK_A, the Boolean expression indicating
the absence of an antecedent failure }
$rd \leftarrow ReadWithDefault(mem, c, a, d)$
$OK_A \leftarrow OK_A \wedge (c \Rightarrow [rd = d])$

Figure 6. Implementation of the STE procedure *AssertMem*.

Procedure *AssertMem* uses the function *ReadWithDefault* in order to assert location *a* of *mem* to data *d* under the contexts *c*. OK_A maintains the condition under which the asserted value is consistent with the current state of the memory. In the case of procedure *CheckMem*, OK_C uses the Boolean expression *found* in order to maintain the condition under which *mem* has data *d* in location *a* given contexts *c*.

procedure *CheckMem*(**List** *mem*, **BExpr** *c*, **AExpr** *a*, **DExpr** *d*)
{ Determine conditions under which location *a* was checked to have data *d*
given contexts *c*, and reflect them on OK_C, the Boolean expression
indicating the absence of a consequent failure}
rd ← *d*
found ← **false**
for each ⟨*ec*, *ea*, *ed*⟩ **in** *mem* from head to tail **do**
match ← *ec* ∧ *a*=*ea*
rd ← *ITE*(*match*, *ed*, *rd*)
found ← *found* ∨ *match*
$OK_C \leftarrow OK_C \wedge (c \Rightarrow [found \wedge rd{=}d])$

Figure 7. Implementation of the STE procedure *CheckMem*.

7. Experimental Results

Experiments were performed on the pipelined addressable accumulator shown in Figure 8. One mode of operation of the circuit is that of initialization of the register file with data from the input `In`, through the adder, and then through the `Hold` register. For this purpose the `Clear` signal is set to 1, so as to clear the value at the second input of the adder, while the destination location in the register file is specified by the address input `Addr`. A second mode of operation of the circuit is that of accumulation. Then, the address input `Addr` specifies a location in the register file, whose contents is to be added to the value supplied at the input `In`. In this case the `Clear` signal is set to 0, so as to ensure that the data value from the output of the register file will be passed unchanged to the adder.

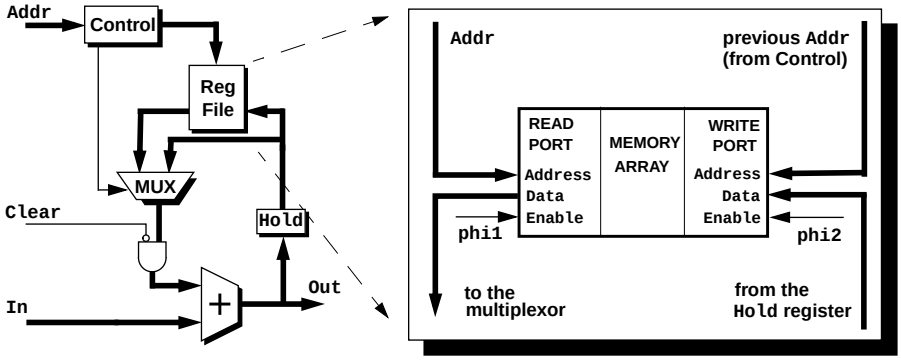


Figure 8. (a) The pipelined addressable accumulator; (b) the connections of its register file when replaced by an EMM. The thick lines indicate buses, while the thin ones are of a single bit.

In order to speed up the accumulation mode by avoiding the latency of the register file, the addressable accumulator is pipelined by the introduction of a `Hold` register, a multiplexor (with the ability to choose between the outputs of the register file and the `Hold` register), and some extra circuitry in the control logic. This extra circuitry consists of a register to store the previous address and a comparator to determine whether that address is identical with the current address at the `Addr` input. Should the two addresses match, the control signal of the multiplexor is set so as to select the output of the `Hold` register. Hence, a bypassing of the register file takes effect.

For the experiments with the EMM, the dual-ported register file is removed from the circuit. The software interface ensures that a *Read* operation takes place on `phi1` and a *Write* operation takes place on `phi2`, according to the register file connections shown in Figure 8.(b).

The specifications necessary for verifying the pipelined addressable accumulator, are presented in (4), (5), and (6). Note that `Reg[`i`]` and `Reg[`j`]` in (5) and (6), respectively, are instances of *symbolic indexing* [1], which results in the total number of symbolic variables being logarithmic in the number of address locations. We construct the antecedents by first defining the operation of the clocks. Shorthand notation for the possible signals applied to the clocks is presented next:

$$Clk01 \doteq (\texttt{phi1} = 0) \wedge (\texttt{phi2} = 1)$$
$$Clk00 \doteq (\texttt{phi1} = 0) \wedge (\texttt{phi2} = 0)$$
$$Clk10 \doteq (\texttt{phi1} = 1) \wedge (\texttt{phi2} = 0)$$

The clocking behavior of the entire circuit over 4, 8, and 12 time periods, respectively, is described by:

$$Clocks_4 \doteq Clk01 \wedge \mathbf{N}(Clk00) \wedge \mathbf{N}^2(Clk10) \wedge \mathbf{N}^3(Clk00)$$
$$Clocks_8 \doteq Clocks_4 \wedge \mathbf{N}^4(Clocks_4)$$
$$Clocks_12 \doteq Clocks_4 \wedge \mathbf{N}^4(Clocks_4) \wedge \mathbf{N}^8(Clocks_4)$$

The first assertion (4) verifies that the `Hold` register can be initialized with data from the input `In` of the pipelined addressable accumulator. Namely, if the `Clear` signal is high, the `Addr` input has an address expression i, and the input `In` has a data expression a, then the output `Out` of the adder will get the data expression a, and so will the `Hold` register, according to the timing details of the implementation (see the timing diagram on Figure 9).

$$Clocks_8 \wedge \mathbf{N}^2((\texttt{Clear} = 1) \wedge (\texttt{Addr} = i) \wedge (\texttt{In} = a)) \overset{\text{LEADSTO}}{\Rightarrow} \mathbf{N}^4(\texttt{Out} = a) \wedge \mathbf{N}^5(\texttt{Hold} = a) \quad (4)$$

The second assertion (5) verifies the adder in the pipelined addressable accumulator. It checks that if the `Addr` input has an address expression k and later, according to the timing details of the implementation, an address expression i, such that then the `Clear` signal is low, and the input `In` has a data expression a, the result will be that the output `Out` of the adder will get the data expression $a + b$, and so will the `Hold` register. Note that the `Hold` register is asserted to data expression b conditionally on

the address equality $i == k$, and that location i of the register file is also asserted to data expression b, however conditionally on the address inequality i != k. If the control logic works properly, it should set the control signal of the multiplexor so as to select the data from the `Hold` register in the event that $i == k$ in order to bypass the register file. Otherwise, the data from location i of the register file will be selected. Altogether, the output of the multiplexor will be equal to $ITE(i == k, b, b) = b$, which will be the data expression at the second adder input. The timing diagram for this assertion can be seen on Figure 9.

$$\begin{aligned} &\mathit{Clocks_12} \wedge \mathbf{N}^2(\texttt{Addr} = k) \wedge \mathbf{N}^5(i == k \rightarrow \texttt{Hold} = b) \wedge \\ &\mathbf{N}^6((\texttt{Clear} = 0) \wedge (\texttt{Addr} = i) \wedge (\texttt{In} = a) \wedge (i \mathrel{!}= k \rightarrow \texttt{Reg}[i] = b)) \\ &\overset{\text{LEADSTO}}{\Rightarrow} \mathbf{N}^8(\texttt{Out} = a + b) \wedge \mathbf{N}^9(\texttt{Hold} = a + b) \end{aligned} \tag{5}$$

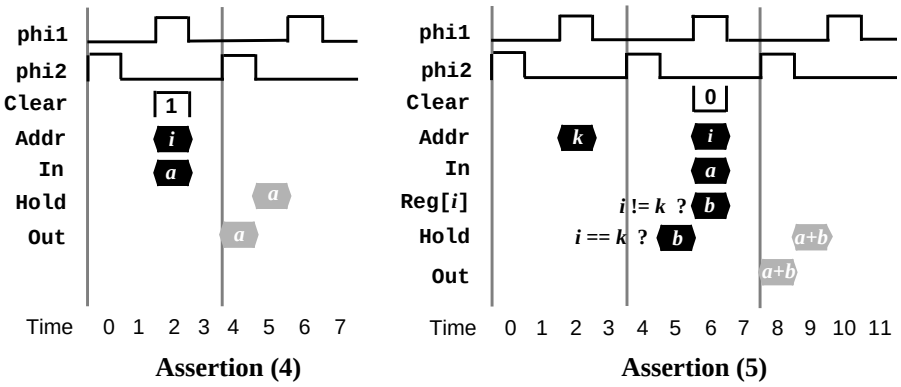


Figure 9. Timing diagrams for assertions (4) and (5). The solid areas denote asserted signals, while the shaded ones represent the expected results.

The last assertion (6) verifies that the register file can maintain its state in the pipelined addressable accumulator. If the `Addr` input has an address expression k and later, an address expression i, such that then a different location j of the register file has data expression b, then the data expression at that location will remain unchanged. The value b in the `Hold` register, asserted conditionally on $j == k$, allows testing the bus from the `Hold` register to the register file by using the same check of the memory state.

$$\begin{aligned} &\mathit{Clocks_12} \wedge \mathbf{N}^2(i \mathrel{!}= j \rightarrow \texttt{Addr} = k) \wedge \mathbf{N}^5((i \mathrel{!}= j \wedge j == k) \rightarrow \texttt{Hold} = b) \wedge \\ &\mathbf{N}^6((i \mathrel{!}= j \rightarrow \texttt{Addr} = i) \wedge ((i \mathrel{!}= j \wedge j \mathrel{!}= k) \rightarrow \texttt{Reg}[j] = b)) \\ &\overset{\text{LEADSTO}}{\Rightarrow} \mathbf{N}^{10}(i \mathrel{!}= j \rightarrow \texttt{Reg}[j] = b) \end{aligned} \tag{6}$$

The experiments were performed on an IBM RS/6000 58H running AIX 4.1.3 with 512 MB of physical memory. As can be seen from Table 1, the EMM outper-

forms the transistor level model (TLM) of the memory array in the pipelined addressable accumulator. A 7-15x speedup and a 2-8x reduction in memory were obtained, with the EMM advantage increasing with the memory size.

# Addresses	# Data Bits	CPU Time (s)			Memory (MB)		
		TLM	EMM	TLM / EMM	TLM	EMM	TLM / EMM
16	16	557	81	6.9	4.2	2.2	1.9
	32	1 095	161	6.8	7.3	3.2	2.3
	64	2 188	315	6.9	13.6	5.2	2.6
	128	4 391	628	7.0	26.3	9.2	2.9
32	16	1 030	100	10.3	8.2	3.0	2.7
	32	2 048	195	10.5	15.3	4.7	3.3
	64	4 102	388	10.6	29.5	8.2	3.6
	128	8 278	781	10.6	57.7	15.2	3.8
64	16	1 992	144	13.8	16.0	4.5	3.6
	32	3 999	283	14.1	30.7	7.8	3.9
	64	7 924	566	14.0	59.8	8.3	7.2
	128	15 824	1 154	13.7	118.0	15.3	7.7
128	16	3 907	248	15.8	31.6	4.6	6.9
	32	7 923	496	16.0	61.6	7.9	7.8
	64	15 547	1 003	15.5	121.1	14.5	8.4
	128	31 079	2 031	15.3	241.7	27.6	8.8

Table 1. Experimental results.

The asymptotic growth of STE, when used together with the TLM and the EMM, is summarized in Table 2, which also does a comparison with symbolic model checking, combined with either a partitioned transition relation [4] or with abstraction functions [6].

Criterion	Symbolic Model Checking		STE	
	Partitioned Transition Relation	Abstraction Functions	TLM	EMM
CPU Time w.r.t. # Data Bits	quadratic	linear	linear	linear
CPU Time w.r.t. # Addresses	cubic	linear	linear	sublinear
Memory w.r.t. # Data Bits	linear	---	linear	sublinear
Memory w.r.t. # Addresses	subcubic	---	linear	sublinear

Table 2. Asymptotic growth comparison of symbolic model checking and STE when verifying simple pipelined data paths.

However, it should be pointed out that there is a slight difference in the pipelined data path used for the experiments in [4] and [6], as compared with the one used in this paper. Also, the memory requirements of symbolic model checking combined with abstraction functions were not reported in [6].

Hence, the new method for efficient modeling of memory arrays has proven to be extremely promising. It would enable the symbolic simulation of memory arrays far larger than previously possible.

8. Future Work

We plan to improve the EMM software interface by including mechanisms to monitor the assumptions for correct operation of the model and to guarantee that it would behave as a conservative approximation of the replaced memory array. Furthermore, we will examine the integration of the efficient memory model with the symmetry-based technique for verification of transistor-level memory arrays, proposed by Pandey and Bryant [7], as a step towards hierarchical verification of systems containing large embedded memories.

Furthermore, we plan to extend the approach in order to support verification methodologies based on comparing the effect that two execution sequences have on the state of a memory array, similar to the work by Burch and Dill [5]. In other words, given two sequences of memory operations, we wish to test whether they yield identical behaviors. The assumption is that the two sequences start with matching initial memory states. For each externally visible *Read* operation in the first sequence, its counterpart in the second sequence must return the same value. Also, the final states resulting from the two sequences must match. To implement this, we require both a mechanism for guaranteeing that consistent values are used for the initial contents of the two memories and an algorithm for comparing the contents of two memories.

References

[1] D. L. Beatty, R. E. Bryant, and C.-J. H. Seger, "Synchronous Circuit Verification by Symbolic Simulation: An Illustration," *Sixth MIT Conference on Advanced Research in VLSI*, 1990, pp. 98-112.

[2] R. E. Bryant, D. E. Beatty, and C.-J. H. Seger, "Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation," *28th Design Automation Conference*, June, 1991, pp. 297-402.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," *27th Design Automation Conference*, June, 1990, pp. 46-51.

[4] J. R. Burch, E. M. Clarke, and D. E. Long, "Representing Circuits More Efficiently in Symbolic Model Checking," *28th Design Automation Conference*, June, 1991, pp. 403-407.

[5] J. R. Burch, and D. L. Dill, "Automated Verification of Pipelined Microprocessor Control," *CAV '94*, D. L. Dill, ed., LNCS 818, Springer-Verlag, June, 1994, pp. 68-80.

[6] E. M. Clarke, O. Grumberg, and D. E. Long, "Model Checking and Abstraction,"*19th Annual ACM Symposium on Principles of Programming Languages*, 1992, pp. 343-354.

[7] M. Pandey, and R. E. Bryant, "Exploiting Symmetry When Verifying Transistor-Level Circuits by Symbolic Trajectory Evaluation," *CAV '97*, June, 1997.

[8] C.-J. H. Seger, and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, Vol. 6, No. 2 (March, 1995), pp. 147-190.