Incorporating Timing Constraints in the Efficient Memory Model for Symbolic Ternary Simulation[1]

Miroslav N. Velev[*]
mvelev@ece.cmu.edu
http://www.ece.cmu.edu/~mvelev

Randal E. Bryant[‡, *]
randy.bryant@cs.cmu.edu
http://www.cs.cmu.edu/~bryant

[*]Department of Electrical and Computer Engineering
[‡]School of Computer Science
Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

Abstract

This paper introduces the four timing constraints of setup time, hold time, minimum delay, and maximum delay in the Efficient Memory Model (EMM). The EMM is a behavioral model, where the number of symbolic variables used to characterize the initial state of the memory is proportional to the number of distinct symbolic memory locations accessed. The behavioral model provides a conservative approximation of the replaced memory array, while allowing the address and control inputs of the memory to accept symbolic ternary values. If a circuit has been formally verified with the behavioral model, the system is guaranteed to function correctly with any memory implementation whose timing parameters are bounded by the ones used in the verification.

1. Introduction

Decreasing feature sizes in new semiconductor technologies increase the wire delays and mandate that simulation and verification with accounting for timing requirements become the norm in the near future. Ternary simulation, where the "unknown" value X is used to indicate that a signal can be either 0 or 1, has long been used to validate digital circuits and to detect timing errors in them [8]. Namely, the value X has been used to represent a signal in transition from one binary value to another. Additionally, X's can be used to model uninitialized nets or "don't-care" conditions. The effects of the X's are propagated through the circuit by the simulator. The use of X's to represent transitions allows the simulator to detect combinational hazards, critical races, and feedback oscillations. Given that the simulation algorithm satisfies a monotonicity property, any binary values resulting when simulating patterns with X's would also result when the X's are replaced by any combination of 0's and 1's [12]. Hence, employing X's reduces the number of simulation patterns, often dramatically. However, ternary simulators will sometimes produce a value X, when an exhaustive analysis would determine the value to be binary (i.e., 0 or 1).

This problem has been resolved by combining ternary modeling with symbolic simulation [1], such that the signals can accept symbolic ternary values, instead of the scalar values 0, 1, and X. Each symbolic ternary value is represented by a pair of symbolic Boolean expressions, defined over a set of symbolic Boolean variables, that encode the cases when the signal would evaluate to 0, 1, or X. The advantage of symbolic ternary simulation is that it efficiently covers a wide range of circuit operating conditions with a single symbolic simulation pattern that involves far fewer variables than would be required for a complete binary symbolic simulation. In addition to validation, symbolic ternary simulation has proven to be very powerful for formal verification, as demonstrated by the Symbolic Trajectory Evaluation (STE) technique [12][7]. Furthermore, symbolic ternary simulation can be combined with different delay models. This has been achieved by Seger and Bryant [11] by assuming that gates have zero delays and are connected in series with delay boxes that model inertial delay bounded by a minimum and a maximum value. However, the application of symbolic simulation has been traditionally restricted to circuits with small memory arrays.

The normal simulation models for memory arrays at the transistor, gate, and behavioral 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. Furthermore, bit-level symbolic model checking [4][5] would need two symbolic variables per memory bit, in order to build the transition relation. Therefore, in both methods the number of variables is proportional to the size of the memory, and is prohibitive for large memory arrays.

This limitation is overcome in our previous work [13] 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 distinct symbolic memory locations accessed rather than to the size of the memory. It is based on the observation that a single execution sequence typically accesses only a limited number of distinct symbolic locations. While the EMM presented in [13] assumes simulation over symbolic binary values, our later work [14] allows the EMM to accept symbolic ternary values at its address and control inputs, while providing a conservative approximation of the replaced memory array. Conservative approximation means that false positive verification results are guaranteed not to occur, although false negative verification

1. This research was supported in part by the SRC under contract 98-DC-068.

results are possible.

The contributions of this paper are that it incorporates the timing parameters of setup time (T_{setup}), hold time (T_{hold}), minimum delay (T_{Dmin}), and maximum delay (T_{Dmax}) into the ternary EMM [14]. Experimental results were obtained using the STE technique.

A symbolic representation of memory arrays has been used by Burch and Dill [6]. They apply uninterpreted functions with equality, which abstract away the details of the data path and allow them to introduce only a single symbolic variable to denote the initial state of the entire memory. Each *Write* or *Read* operation results in building a formula over the current memory state, so that the latest memory state is a formula reflecting the sequence of memory writes. In our method, the memory state is represented with a list of entries encoding the sequence of updates of symbolic addresses with symbolic data. Our *Write* operation modifies this list. However, we perform the verification at the circuit level of the implementation and need bit-level data for symbolic word-level memory locations in order to verify the data path. This requires the user to introduce symbolic variables proportional to both the number of distinct symbolic memory locations accessed and the number of data bits per location.

The introduction of minimum and maximum delays for memory read ports in our model is conceptually similar to the way that Seger and Bryant [11] model inertial delay bounded by a minimum and a maximum value, although implemented within the software interface that links the EMM and the rest of the circuit. However, our treatment of setup and hold times for memory ports is tailored to the specifics of the EMM in the context of symbolic ternary simulation. Namely, setup and hold times are accounted for by means of an auxiliary circuit that translates timing violations into ambiguity expressed by the value X at the control input of the corresponding memory port. The design of the auxiliary circuit, the EMM, and its software interface guarantee a conservative approximation of the replaced memory array.

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 [9][10]. The second step is to use STE to verify the circuit after the memory arrays are replaced by EMMs and is the focus of the present work.

In the remainder of the paper, Section 2 describes the symbolic domains used in our algorithms. Section 3 presents the EMM and a way to incorporate minimum and maximum delays in the model. Section 4 explains the EMM algorithms. Section 5 shows how to incorporate setup and hold times in the model. Experimental results and conclusions are presented in Section 6, and plans for future work are outlined in Section 7.

2. Symbolic Domains

We will consider three different domains - control, address, and data - corresponding to the three different types of information that can be applied at the inputs of a memory array. A control expression c will represent the value of a node in ternary symbolic simulation and will have a high encoding $c.h$ and a low encoding $c.l$, each of which is a Boolean expression. The ternary values that can be represented by a control expression c are shown in Table 1. We would write $[c.h, c.l]$ to denote c. It will be assumed that $c.h$ and $c.l$ cannot be simultaneously **false**. The types **BExpr**, **CExpr** will denote respectively Boolean and control expressions in the algorithms to be presented.

Ternary value	$c.h$	$c.l$
0	**false**	**true**
1	**true**	**false**
X	**true**	**true**

Table 1. 2-bit encoding of ternary logic.

The memory address and data inputs, since connected with circuit nodes, will receive ternary values represented as control expressions. Hence, addresses and data will be represented by vectors of control expressions having width n and w, respectively, for a memory with $N = 2^n$ locations, each holding a word consisting of w bits. Observe that an X at a given bit position represents the "unknown" value, i.e., the bit can be either 0 or 1, so that many distinct addresses or data will be represented. To capture this property of ternary simulation, we introduce the type **ASExpr** (address set expression) to denote a set of addresses. Similarly, the type **DSExpr** (data set expression) will denote a set of data. Note that in both cases, a set will be represented by a single vector of ternary values. We will use the notation $\langle a_1, \ldots, a_n \rangle$ to explicitly represent the address set expression a, where a_i is the control expression for the corresponding bit position of a. Data set expressions will have a similar explicit representation, but with w bits. Symbolic variables will be introduced in each of the domains and will be used in expression generation.

The symbol $\mathcal{U}_{\mathcal{D}}$ will designate the universal data set. It will represent the most general information about a set of data. In ternary logic, $\mathcal{U}_{\mathcal{D}}$ can be modeled by a vector of control expressions consisting entirely of X's.

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**.

A symbolic predicate is a function which takes symbolic arguments and returns a symbolic Boolean expression. The following symbolic predicates will be used in our algorithms, where c is of type **CExpr**, and a is of type **ASExpr**:

$$Zero(c) \doteq \neg c.h \wedge c.l, \tag{1}$$

$$Hard(c) \doteq c.h \wedge \neg c.l, \tag{2}$$

$$Soft(c) \doteq c.h \wedge c.l, \tag{3}$$

$$Unique(a) \doteq \bigwedge_{i=1}^{n} \neg Soft(a_i). \tag{4}$$

The predicates *Zero*, *Hard*, and *Soft* define the conditions for their arguments to be the ternary 0, 1, and X, respectively. The

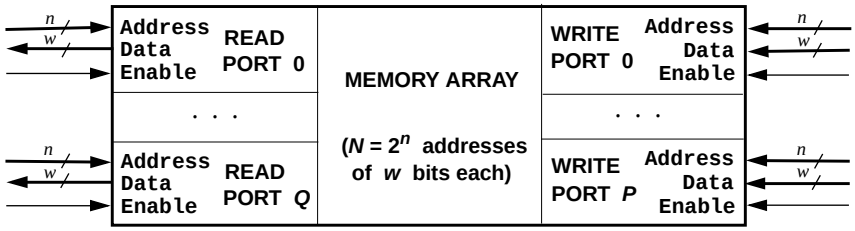


Figure 1. View of a memory array, according to our model.

predicate *Unique* defines the condition for the address set expression a to represent a *unique* or single address.

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

$$ITE(b,\ t,\ e)\ \doteq\ (b \wedge t) \vee (\neg b \wedge e). \quad (5)$$

Address set comparison with another address set is implemented as:

$$a_1 = a_2\ \doteq\ \neg \bigvee_{i=1}^{n} [(a_1.h_i \oplus a_2.h_i)\ \vee\ (a_1.l_i \oplus a_2.l_i)], \quad (6)$$

where $a_1.h_i$ and $a_1.l_i$ represent the high and low encodings of the control expression for bit i of address set expression a_1. Address set selection $a_1 \leftarrow ITE(b,\ a_2,\ a_3)$ is implemented by selecting the corresponding bits:

$$a_1.h_i \leftarrow ITE(b,\ a_2.h_i,\ a_3.h_i),$$

$$a_1.l_i \leftarrow ITE(b,\ a_2.l_i,\ a_3.l_i),\quad i = 1, \ldots, n. \quad (7)$$

Checking whether address set a_1 is a subset of address set a_2 is done by:

$$a_1 \subseteq a_2\ \doteq\ \neg \bigvee_{i=1}^{n} (a_1.h_i \wedge \neg a_2.h_i\ \vee\ a_1.l_i \wedge \neg a_2.l_i), \quad (8)$$

and checking address sets a_1 and a_2 for overlap is implemented by:

$$Overlap(a_1, a_2)\ \doteq\ \bigwedge_{i=1}^{n} (a_1.l_i \wedge a_2.l_i\ \vee\ a_1.h_i \wedge a_2.h_i). \quad (9)$$

The definition of symbolic predicates over data set expressions is similar, but over vectors of width w.

Note that all of the above predicates are symbolic, i.e., they return a symbolic Boolean expression and will be true in some contexts and false in others. Therefore, a symbolic predicate cannot be used as a control decision in algorithms. The function *Valid*, when applied to a symbolic Boolean expression, will return **true** if the expression is valid or equal to **true** (i.e., true for all contexts), and will return **false** otherwise. We can make control decisions based on whether or not an expression is valid.

We will also need to form a data set expression which is the union of two data set expressions, d_1 and d_2. If these differ in exactly one bit position, i.e., one of them has a 0 and the other a 1, then the ternary result will have an X in that bit position and will be an exact computation. However, if d_1 and d_2 differ in many bit positions, these will be represented as X's in the ternary result and that will not always yield an exact computation. For example, if $d_1 = \langle 0,\ 1\rangle$ and $d_2 = \langle 1,\ 0\rangle$, the result will be $\langle$X, X$\rangle$ and will not be exact, as it will also contain the data set expressions $\langle 0,\ 0\rangle$ and $\langle 1,\ 1\rangle$, which are not subsets of d_1 or d_2. We define the operation *approximate union* $d_1 \mathbin{\tilde{\cup}} d_2$ of two data set expressions as:

$$[d_1 \mathbin{\tilde{\cup}} d_2\ ]_i\ \doteq\ [d_1.h_i \vee d_2.h_i,\ d_1.l_i \vee d_2.l_i],\quad i = 1, \ldots, w. \quad (10)$$

We have used Ordered Binary Decision Diagrams (BDDs) [3] to represent the Boolean expressions in our implementation. However, there is nothing about this work that intrinsically requires it to be BDD based. Any canonical representation of Boolean expressions can be substituted.

3. 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.

The interaction of the memory array with the rest of the circuit is assumed to take place when a port `Enable` signal is not 0. In case of multiple port `Enable`s not being 0 simultaneously, the resulting accesses to the memory array will be ordered according to the priority of the ports.

Write ports can have requirements for T_{setup} and T_{hold}, while read ports can additionally have requirements for T_{Dmin} and T_{Dmax}. The port inputs must be stable from T_{setup} units of time before the rising edge of the port `Enable` until T_{hold} units of time after that in order for the memory operation to take place correctly. For the same reason, we require that the port `Enable` be stable for T_{hold} units of time after its rising edge. T_{hold} is the time necessary to correctly access the memory location, specified by the port address inputs, after the port `Enable` signal goes high. In the case of read ports, T_{Dmin} and T_{Dmax} give the min. and max. delays, respectively, needed for the data to propagate from the memory storage cells to the port data outputs. These four timing parameters can take only non-negative integer values with $T_{Dmin} \le T_{Dmax}$.

During symbolic simulation, the memory state is represented by a list containing entries of the form $\langle h, s, a, d\rangle$, where h and s are Boolean expressions denoting the set of contexts for which the entry is defined, a is an address expression denoting a memory location, and d is a data expression representing the contents of this location. The context information is included for modeling memory systems where the *Write* operation may be performed conditionally on the value of a control signal c. The Boolean expression h represents the contexts $Hard(c) \wedge Unique(a)$, when the control signal was 1 and the

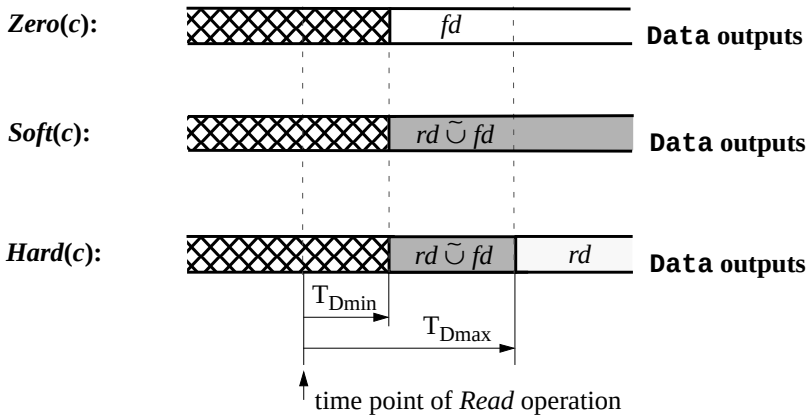


Figure 2. Timing diagrams for the data that will appear at the Data outputs of a read port, under the three possible contexts for the value *c* of the port's Enable signal. *rd* is the data set expression retrieved from memory by the *Read* operation. *fd* is the data that will appear at the Data outputs T_{Dmin} units of time ahead from the *Read* operation time, if the *Read* did not take place.

address *a* was unique. Under contexts *h* the location *a* is definitely overwritten with data *d*. The Boolean expression *s* represents the contexts $Soft(c) \vee Hard(c) \wedge \neg Unique(a)$, when the control signal was an X, or it was a 1 and the address was not unique. Under contexts *s* the location *a* is uncertainly overwritten with data *d*. Initially the list is empty. The type **List** will be used to denote such memory lists.

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 memory input lines. Should a memory input value change, given that its corresponding port `Enable` value *c* is not 0, a *Write* or a *Read* operation will result, as determined by the type of the port. The `Address` and `Data` lines of the port will be scanned in order to form the address set expression *a* and the data set expression *d*, respectively. A *Write* operation takes as arguments both *a* and *d*, while a *Read* operation takes only *a*. Both of these operations will be presented in the next section.

A *Read* operation retrieves from the list a data set expression *rd* that represents the data contents of address *a*. The software interface completes the read by scheduling the `Data` lines of the port to be updated with the data set expression $ITE(Hard(c), rd, ITE(Soft(c), (rd \uplus fd), fd))$ at T_{Dmax} units of time ahead of the current simulation time. The data set expression *fd* is the one that the `Data` lines will otherwise have in that future simulation time interval. Additionally, in the case when $T_{Dmin} < T_{Dmax}$, the software interface schedules the `Data` lines of the read port to be updated with the data set expression $ITE(Zero(c), fd, (rd \uplus fd))$ at T_{Dmin} units of time ahead of the current simulation time. Again, the data set expression *fd* is the one that will otherwise appear at the `Data` outputs in that future simulation time. This implementation of minimum and maximum delays in the EMM preserves its behavior as a conservative approximation of the replaced memory array.

Figure 2 illustrates how a *Read* operation will affect the values of the port `Data` outputs. Under the contexts where the value *c* of the port `Enable` is 0, the data at the port `Data` lines will be unchanged and will be *fd*. Under the contexts when *c* is a 1, the approximate union $rd \uplus fd$, where *rd* is the data set expression retrieved by the *Read* operation, will be the value of the `Data` outputs in the interval from T_{Dmin} to T_{Dmax} ahead from the read time, to be followed by a value of *rd* after that. Finally, under the contexts when *c* is an X, $rd \uplus fd$ will appear at the `Data` lines after a delay of T_{Dmin}.

After completing a *Write* operation, the software interface checks every read port of the same memory for a possible ongoing read (as determined by the read port `Enable` value being different from 0) from an address that overlaps the one of the recent write. For any such port, a *Read* operation is invoked immediately and the `Data` lines of the read port are updated with $rd \uplus d$ where *rd* is the data set expression returned by the *Read*, and *d* is the data set expression that would otherwise appear on the `Data` lines at that time. This guarantees that the EMM would behave as a conservative approximation of the replaced memory array.

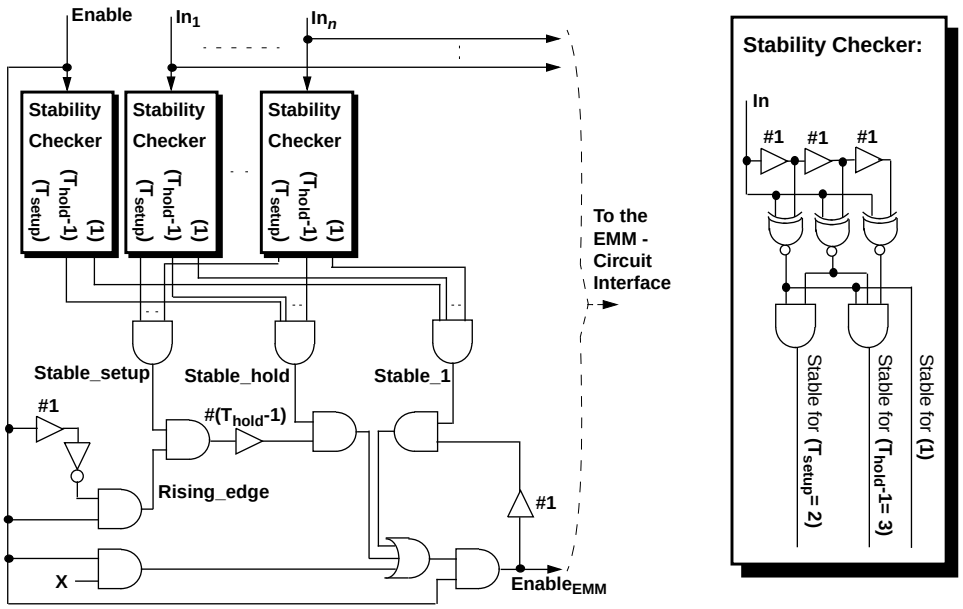
4. Implementation of Memory Operations

4.1 Support Operations

The list entries are kept in order from *head* (low priority) to *tail* (high priority). The initial state of every memory location is assumed to contain arbitrary data and is represented with the universal data set $\mathcal{U}_{\mathcal{D}}$. Entries in the list from low to high priority model the sequence of memory writes with the tail entry being the result of the latest memory update. Entries may be inserted at the tail end only, using procedure *InsertTail*, and may be deleted using procedure *Delete*.

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

The *Write* operation, shown as a procedure in Figure 3, takes as arguments a memory list, a control expression denoting



(a) **(b)**

Figure 5. (a) The auxiliary circuit incorporating setup and hold times for one port of an EMM-modeled memory array; (b) implementation of the stability checker for T_{setup} = 2 and T_{hold} = 4. The buffers have integer delays denoted by the numbers after the # sign.

the contexts for which the write should be performed, and address set and data set expressions denoting the memory location and its desired contents, respectively. As the code shows, the write 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.

procedure *Write*(**List** *mem*, **CExpr** *c*, **ASExpr** *a*, **DSExpr** *d*)
/* Write data *d* to location *a* under control *c* */
  $h \leftarrow Hard(c) \wedge Unique(a)$
  $s \leftarrow Soft(c) \ \vee \ Hard(c) \wedge \neg Unique(a)$
  InsertTail(*mem*, $\langle h, s, a, d \rangle$)

Figure 3. Implementation of the *Write* operation.

The *Read* operation is shown in Figure 4 as a function which, given a memory list and an address set expression, returns a data set expression indicating the contents of this location. It does so by scanning through the list from lowest to highest priority. For each list entry, a Boolean expression *hard_match* is built that indicates the contexts under which the entry is hard (definite) and its (unique) address equals the read address *a*. Under these contexts, that element's data *ed* is selected. Else, under the contexts expressed by the Boolean expression *soft_match*, the approximate union of the element's data and the previously formed data is selected. Finally, under the contexts when both *hard_match* and *soft_match* are false, the previously formed data is kept.

function *Read*(**List** *mem*, **ASExpr** *a*) : **DSExpr**
/* Attempt to read from location *a* */
  $rd \leftarrow \mathcal{U}_{\mathcal{D}}$
  if $\neg Valid(\neg Unique(a))$ **then**
    for each $\langle eh, es, ea, ed \rangle$ **in** *mem* from head to tail **do**
      $hard\_match \leftarrow eh \wedge (ea = a)$
      $soft\_match \leftarrow (es \ \vee \ eh \wedge \neg(ea = a)) \wedge Overlap(ea, a)$
      $rd \leftarrow ITE(hard\_match, ed, ITE(soft\_match, (ed \ \tilde{\cup} \ rd), rd))$
    return *rd*

Figure 4. Implementation of the *Read* operation.

$\mathcal{U}_{\mathcal{D}}$ is used as the default data set expression. The contexts for which *Read* does not find a matching address in the list are those for which the addressed memory location has never been accessed by a write. The data set expression $\mathcal{U}_{\mathcal{D}}$ is then returned to indicate that the location may contain arbitrary data.

The *Read* operation is designed to be precise only in the contexts when the argument address set expression is unique, and to return $\mathcal{U}_{\mathcal{D}}$ otherwise. The expression *soft_match* is defined so that for any list entry, whose address intersects the read address *a*, the approximate union of the entry's data set expression and the previously formed data set expression is selected. Note that in the contexts when the currently examined list element is hard, as determined by *eh*, we require that the element's address does not equal the read address (so that it is a proper subset of it). This ensures that the Boolean expressions for *hard_match* and *soft_match* will not be true simultaneously. For an implementation of the *Read* operation that yields more precise results, and

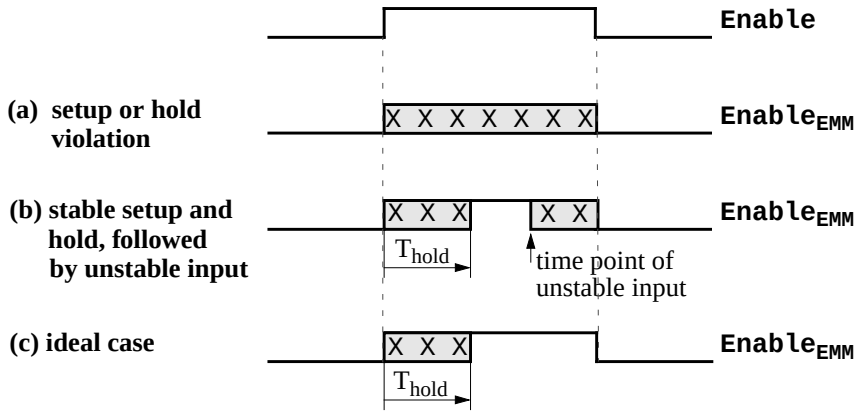


Figure 6. Timing diagrams for signal $\texttt{Enable}_{\texttt{EMM}}$ in the case of three scenarios for the inputs of the auxiliary circuit: (a) setup or hold violation; (b) stable setup and hold, followed by an unstable input, including Enable; (c) stable setup and hold, followed by stable inputs until the falling edge of Enable - the ideal case.

for an optimized version of the *Write* operation, the reader is referred to [14].

5. Incorporation of Setup and Hold Times into the EMM

To detect errors due to timing violations we adopt the following philosophy. Rather than flag the violation as a verification failure, we model the potential data corruption caused by the violation with patterns containing X values. Subsequent operations will further propagate these corrupted values, causing verification failures when the final state is checked. By this means we avoid "false negative" verification failures, e.g., when a timing violation occurs under a "don't-care" condition.

Setup and hold times can be introduced for each port of the EMM-modeled memory array by means of the auxiliary circuit shown in Figure 5. $\texttt{Enable}_{\texttt{EMM}}$ is the port enable signal supplied to the EMM-circuit interface. This circuit detects potential timing violations and translates them into X values on $\texttt{Enable}_{\texttt{EMM}}$. The inputs $\texttt{In}_1$ through $\texttt{In}_n$ in the figure include all the port inputs except for the Enable, i.e., both the Address and Data lines in the case of a write port, and only the Address lines in the case of a read port. The auxiliary circuit assumes level-sensitive memory operations, so that T_{hold} is required to be greater than or equal to 1. If only one of the conditions $T_{setup} = 0$ or $T_{hold} = 1$ is satisfied, the corresponding output of the stability checker can be a wire connected to 1. If both $T_{setup} = 0$ and $T_{hold} = 1$, then $\texttt{Enable}_{\texttt{EMM}}$ can be connected directly to Enable.

Observe that $\texttt{Enable}_{\texttt{EMM}}$ goes to 0 or to X together with Enable. However, when Enable goes from 0 to 1, $\texttt{Enable}_{\texttt{EMM}}$ will become an X, which will be replaced by a 1 as soon as the setup and hold requirements are satisfied. By this means we can detect an error caused by a read that overlaps the time period between the beginning of the write operation, as determined by the Enable of the write port making a 0→1 transition, and before T_{hold} for the write has elapsed.

The timing diagrams in Figure 6 illustrate the signal $\texttt{Enable}_{\texttt{EMM}}$ in the case of several scenarios for the inputs of the auxiliary circuit. If the setup or hold requirements are violated, then $\texttt{Enable}_{\texttt{EMM}}$ will stay at X for the entire period when Enable is high - see Figure 6.(a). If the setup and hold requirements are satisfied, so that $\texttt{Enable}_{\texttt{EMM}}$ is already 1, but then one of the port inputs (including Enable) is unstable, the signal $\texttt{Enable}_{\texttt{EMM}}$ will go to an X, as shown in Figure 6.(b). In the ideal case (see Figure 6.(c)) the auxiliary circuit will trigger two memory operations - the first one due to the 0→X transition, and the second one due to the X→1 transition of $\texttt{Enable}_{\texttt{EMM}}$.

The definition of signal $\texttt{Enable}_{\texttt{EMM}}$ according to the auxiliary circuit of Figure 5, together with the implementation of the EMM and the EMM-circuit interface, guarantees that the EMM will behave as a conservative approximation of the replaced memory array.

If a circuit has been formally verified with the EMM, the definition of the EMM and the EMM-circuit interface guarantee that the system will function correctly with any memory implementation whose timing parameters are bounded by the ones used in the verification. This result means that for a write port with parameters T_{setup} and T_{hold} used in the verification, any memory implementation, where the port has parameters T'_{setup} and T'_{hold}, such that $T'_{setup} \le T_{setup}$, and $T'_{hold} \le T_{hold}$, will be correct. Similarly, for a read port with parameters T_{setup}, T_{hold}, T_{Dmin}, and T_{Dmax}, any implementation where the port's new parameters satisfy $T'_{setup} \le T_{setup}$, $T'_{hold} \le T_{hold}$, and $T_{Dmin} \le T'_{Dmin} \le T'_{Dmax} \le T_{Dmax}$, will function correctly.

6. Experimental Results

Experiments were performed on the pipelined addressable accumulator - a pipelined data path with Execution and Write-Back stages. Forwarding of the result in Write-Back is takes place, if needed. Data values and results are stored in a dual-ported register file, which we replaced with an EMM (assuming it has been verified separately). Experimental results were obtained using the Symbolic Trajectory Evaluation technique

[12] for formal verification. For a description of the circuit and its specifications, the reader is referred to [13].

The experiments were performed on an IBM RS/6000 43P-140 with a 233MHz PowerPC 604e microprocessor, having 512 MB of physical memory, and running AIX 4.1.5. Results for the influence of T_{setup} and T_{hold} on the CPU time and memory needed for the verification of the pipelined addressable accumulator are presented in Table 2. Incorporation of these two timing requirements resulted in a severe penalty, which is partly an artifact of the not very efficient simulation engine in our tool. Namely, the simulation algorithm has a quadratic complexity in terms of the number of circuit nets when processing the events for a single time interval, while an optimal algorithm would have a linear complexity. The difference becomes pronounced when simulating circuits with long dependence chains of gates, as is the case for the stability checkers in the auxiliary circuit that we propose.

T_{setup}	CPU Time [s]				Memory [MB]			
	T_{hold}				T_{hold}			
	0	1	10	100	0	1	10	100
0	207	261	312	986	3.8	3.9	4.1	6.9
1	234	271	313	977	3.9	3.9	4.1	6.9
10	278	321	331	1 009	4.1	4.1	4.1	6.9
100	899	943	966	1 188	6.8	6.8	6.9	6.9

Table 2. Experimental results for the influence of T_{setup} and T_{hold} on the CPU time and memory required for the verification of the pipelined addressable accumulator with a 64×64 Register File, modeled by an EMM.

We also performed experiments for determining the effect of T_{Dmin} and T_{Dmax} on the CPU time and memory needed for the verification of the pipelined addressable accumulator. Choosing $T_{setup} = T_{hold} = 0$, $N = w = 64$, while varying T_{Dmin} in the range of 0 to 1000, and T_{Dmax} in the range of T_{Dmin} to 1000, resulted in a CPU time overhead of less than 4.5% and a memory requirement overhead of less than 0.3%. The extra events processed were negligible in number and made no difference for the event-driven simulation engine, which handles efficiently consecutive events that are widely separated in time. Therefore, incorporation of minimum and maximum delays for read ports of the EMM comes at no extra cost, given a correctly functioning circuit, while enabling simulation and formal verification under timing requirements.

7. Future Work

The efficiency of modeling setup and hold times can be increased significantly if the auxiliary circuit that monitors them is implemented behaviorally, as part of the EMM. Particularly, if ways are provided for the event scheduler to inform the EMM interface that there are no events on the event queue for a certain period ahead, then the behavioral implementation of the auxiliary circuit can advance the state of its delay chains within the stability checkers instantly. Before that, the EMM interface needs to communicate to the event scheduler the maximum number of time steps it can advance. This will enable the event scheduler to coordinate EMMs with different setup and hold times that coexist in the same circuit, by allowing all EMMs to advance their internal state with the minimum interval chosen among those time intervals and the time to the next event on the event queue.

One can also explore ways that would give the user flexibility to define variations of the EMM-circuit interface, according to the timing constraints of a particular memory circuit.

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] R.E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Serveys*, Vol. 24, No. 3 (September 1992), pp. 293-318.

[4] 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.

[5] 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.

[6] 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.

[7] A. Jain, "Formal Hardware Verification by Symbolic Trajectory Evaluation," Ph.D. thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, August 1997.

[8] J.S. Jephson, R.P. McQuarrie, and R.E. Vogelsberg, "A Three-Value Computer Design Verification System," *IBM Systems Journal*, Vol. 8, No. 3 (1969), pp. 178-188.

[9] M. Pandey, "Formal Verification of Memory Arrays," Ph.D. thesis, School of Computer Science, Carnegie Mellon University, May 1997.

[10] M. Pandey, and R.E. Bryant, "Exploiting Symmetry When Verifying Transistor-Level Circuits by Symbolic Trajectory Evaluation," *CAV '97*, O. Grumberg, ed., LNCS 1254, Springer-Verlag, June, 1997, pp. 244-255.

[11] C.-J.H. Seger, and R.E. Bryant, "Modeling of Circuit Delays in Symbolic Simulation," *Formal VLSI Correctness Verification: VLSI Design Methods, II*, L.J.M. Claesen, ed., Elsevier Science Publishers B.V., 1990, pp. 23-37.

[12] 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.

[13] M.N. Velev, R.E. Bryant, and A. Jain, "Efficient Modeling of Memory Arrays in Symbolic Simulation,"[2] *CAV '97*, O. Grumberg, ed., LNCS 1254, Springer-Verlag, June, 1997, pp. 388-399.

[14] M.N. Velev, and R.E. Bryant, "Efficient Modeling of Memory Arrays in Symbolic Ternary Simulation,"[2] *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, B. Steffen, ed., LNCS 1384, Springer-Verlag, March-April, 1998, pp. 136-150.

2. Available from: `http://www.ece.cmu.edu/~mvelev`