Verification of Pipelined Microprocessors by Correspondence Checking in Symbolic Ternary Simulation[1]

Miroslav N. Velev*
mvelev@ece.cmu.edu

Randal E. Bryant‡, *
randy.bryant@cs.cmu.edu

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

Abstract

This paper makes the idea of memory shadowing [5] applicable to symbolic ternary simulation. Memory shadowing, an extension of Burch and Dill's pipeline verification method [6] to the bit level, is a technique for providing on-the-fly identical initial memory state to two different memory execution sequences. We also present an algorithm which compares the final states of two memories for ternary correspondence, as well as an approach for generating efficiently the initial state of memories. These techniques allow us to verify that a pipelined circuit has behavior corresponding to that of its unpipelined specification by simulating two symbolic ternary execution sequences and comparing their final memory states. Experimental results show the potential of the new ideas.

1. Introduction

This paper makes memory shadowing [5] applicable to symbolic ternary simulation. Memory shadowing is a technique for providing on-the-fly identical initial memory state to two different memory execution sequences. Combined with an algorithm which compares the final states of two memories, memory shadowing allows us to verify that a pipelined circuit has behavior corresponding to that of its unpipelined specification. This is done by simulating two symbolic execution sequences and comparing their final memory states. However, the method presented in [5] assumes simulation over symbolic binary values.

Ternary simulation, where the "unknown" value X is used to indicate that a signal can be either 0 or 1, has proven to be more powerful than binary simulation for both validation and formal verification of digital circuits [17]. 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. 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 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.

One of the hurdles in simulation has been the representation of memory arrays. These have been traditionally modeled by explicitly representing every memory bit. While this is not a problem for conventional simulation, symbolic simulation would require a symbolic variable to denote the initial state of every memory bit. Therefore, the number of variables would be proportional to the size of the memory, and would be prohibitive for large memory arrays.

This limitation is overcome in [18] by replacing each memory array with an Efficient Memory Model (EMM), assuming that the memory address and control inputs can accept only symbolic binary values. 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 typical verification execution sequence usually accesses only a limited number of distinct symbolic locations. Memory state is represented in the EMM by a list of entries encoding the relative history of memory operations. The list interacts with the rest of the circuit by means of a software interface developed as part of the symbolic simulator.

A ternary version of the EMM is presented in [19], where the memory address and control inputs are allowed

1. This research was supported in part by the SRC under contract 97-DC-068.

Copyright 1998 IEEE. Published in the Proceedings of CSD'98, March 1998 Fukushima, Japan. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

to accept symbolic ternary values, while the EMM will behave as a conservative approximation to the replaced memory array. Conservative approximation means that false positive verification results are guaranteed not to occur, although false negative verification results are possible. Since symbolic ternary values are a superset of symbolic binary values, the ternary EMM is a superset of the binary EMM from [18], while also exploiting the computational power of symbolic ternary simulation.

Burch and Dill also use a symbolic representation of memory arrays [6]. They apply uninterpreted functions with equality, which allows 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 state of the memory, so that the latest memory state reflects the sequence of memory writes. However, we need bit-level data for various memory locations in order to verify the data path. This requires our algorithms to introduce symbolic variables proportional to both the number of distinct symbolic memory locations accessed and to the number of data bits per location. Furthermore, we need the flexibility to include new symbolic memory locations as part of the initial memory state at any point in the verification process. Hence, the memory state in our case reflects the relative history of memory operations, rather than the sequence of writes. This difference will become clear as we present our algorithms.

The memory shadowing technique [5] is an extension of Burch and Dill's pipeline verification method [6] to a bit-level circuit verification. The correctness criterion of Burch and Dill's method is expressed with a commutative diagram and an underlying abstraction function - see Figure 1. This correctness criterion is not new to verification. It has been introduced by Hoare [8] for verifying computations on abstract data types in software, and has been used by Bose and Fisher [2] to verify pipelined circuits. All these verification methods are based on comparing an implementation transformation F_{Impl} against a specification transformation F_{Spec}. The assumption is that the two transformations start from a pair of matching initial states - Q_{Impl} and Q_{Spec}, respectively - where the match is determined according to some abstraction function *Abs*. The correctness criterion is that the two transformations should yield a pair of matching final states - Q'_{Impl} and Q'_{Spec}, respectively - where the match is determined by the same abstraction function. In other words, the abstraction function should make the diagram commute. Note that there are two paths from Q_{Impl} to Q'_{Spec}. We will refer to the one that involves F_{Impl} as the implementation side of the commutative diagram, while the one that involves F_{Spec} will be called the specification side.

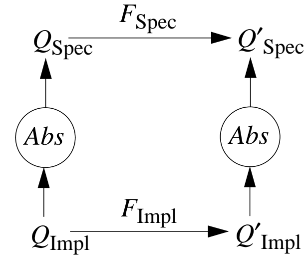

Figure 1. Commutative diagram for the correctness criterion

The Burch and Dill approach is conceptually elegant in the way it uses a symbolic simulation of the hardware design to automatically compute an abstraction function from the pipeline state to the user-visible state. Namely, starting from a general symbolic initial state Q_{Impl} they simulate a *flush* of the pipeline by stalling it for a sufficient number of cycles that will allow all partially executed instructions to complete. Then, they consider the resulting state of the user-visible memory elements (e.g., the register file and the program counter) to be the matching state Q_{Spec}.

Burch and Dill's implementation [6][7] of their method requires a high level abstract model of the implementation that still exposes relevant design issues, such as pipelining. They work on models that completely represent the control path of the processor, but hide the functional details of the data path by means of uninterpreted functions. Our implementation of Burch and Dill's method, presented in this paper, allows verification at the bit level. By verifying at the bit level, we avoid the need to construct an abstracted model of the circuit. We can verify the actual hardware design, given a logic gate-level or register-transfer-level description.

An extensive body of research has been spawned by Burch and Dill's method. Sawada and Hunt [16] have combined it with theorem proving, assuming the availability of a set of invariants that completely specifies the properties of the pipelined processor in correct operation. Burch [7] has extended it to superscalar processor verification by proposing a new flushing mechanism (notice that the method requires an arbitrary abstraction function that makes the correctness criterion diagram commute) and by decomposing the commutative diagram from [6] into three more easily verifiable commutative diagrams. The correctness of this decomposition is proven by Windley and Burch [21]. Jones, Seger, and Dill [12] propose the use of the pipeline as a specification for the correctness of its forwarding logic. They apply two specially designed instruction sequences that should yield identical behaviors and compare their effects on the register file. One of the sequences completely fills the pipeline with instructions and then flushes it with a sequence of NOPs, while the

other consists of the same instructions but separated with as many NOPs as to avoid the exercising of the forwarding logic.

The contributions of this paper are: 1) combining the memory shadowing technique [5] with the ternary EMM [19] in order to support the efficient symbolic ternary simulation when extending Burch and Dill's pipeline verification method [6] to completely model the functionality of the data path at the bit level; 2) a modified correctness criterion in the form of a commutative diagram targeted to symbolic ternary simulation; 3) a variable-group indexing technique for generating the initial state of EMMs; and 4) experimental results with a shortened MIPS pipeline showing the potential of the new ideas.

We consider two forms of verification, supported by our tool: 1) *Symbolic Trajectory Evaluation* (STE) [17], where one proves that a circuit satisfies a specification given as a temporal logic formula; and 2) *Correspondence Checking*, where one proves a correspondence between two circuits by evaluating two execution sequences starting from a common initial state and showing that they yield identical final user-visible states, based on the commutative diagram of Figure 1. We propose using both forms as part of a four step approach for the verification of pipelined processors. The first step is to use STE to verify the transistor-level memory elements (both memory arrays and latches), independently from the rest of the circuit. Pandey and Bryant [13][14] have combined symmetry reductions and STE to enable the verification of very large memory arrays at the transistor level. The second step is to replace the memory arrays with EMMs for both the implementation and the specification circuits. The third step is to use STE to verify the non-pipelined specification circuit, which is assumed to support the same instruction set architecture and to have the same user-visible state as the pipelined processor. One of our previous papers describes the use of the EMM in this context [18]. The fourth and last step is to perform correspondence checking between the pipelined processor and its specification. This step is the focus of the present work.

Another form of verification is symbolic model checking [11]. It uses the transition relation of a system and an initial set of states in order to find the set of reachable states and to verify whether they satisfy certain properties. The temporal logic supported by symbolic model checking is very rich. However, the method requires two symbolic Boolean variables per bit of state in order to build the transition relation. This precludes it from being used for verification of systems with very large state spaces, and makes it most applicable for verification of control intensive systems, such as communication and cache coherence protocols. On the other hand, STE and Correspondence Checking are suitable for verification of data intensive systems. Both methods avoid building a transition function for the system by constructing it on-the-fly during symbolic simulation. This allows them to employ the EMM for representing the state of the system in a very efficient way.

In the remainder of the paper, Section 2 summarizes Burch and Dill's pipeline verification method. Section 3 describes the symbolic domain used in our algorithms. Section 4 presents the assumptions, data structures, and algorithms of the ternary EMM. The memory shadowing technique for providing on-the-fly identical initial memory state to two different ternary execution sequences is explained in Section 5, which also presents an algorithm that compares for correspondence the states of two memory arrays. The variable-group indexing technique for generating the initial state of EMMs is presented in Section 6. The verification methodology for ternary correspondence checking by applying memory shadowing is described in Section 7. Experimental results are summarized in Section 8. Finally, conclusions are drawn and future work is outlined in Section 9.

2. Burch and Dill's Pipeline Verification Method

When verifying a pipelined processor, Burch and Dill [6] assume the availability of a specification non-pipelined circuit, which has user-visible state $U = \{0, 1\}^n$ and input state $I = \{0, 1\}^m$. The implementation (possibly pipelined) circuit is assumed to have the same user-visible and input state, although it can also have pipeline state $P = \{0, 1\}^k$. The combined user-visible and pipeline state of the implementation is then $U \times P$ and will be written in the form $\langle \vec{u}, \vec{p} \rangle$. Each of the circuits is characterized with its transition function $\delta_I : I \times (U \times P) \rightarrow (U \times P)$ for the implementation, and $\delta_S : I \times U \rightarrow U$ for the specification.

Burch and Dill further assume that if the implementation is pipelined, it has or can be modified to include a stall input. When asserted, this input will prevent new instructions from entering the pipeline, while letting partially executed instructions advance and allowing the pipeline state to be flushed. The notation *Stall* will be used for the implementation's transition function when the stall input is asserted. It is also assumed that the two circuits support the same instruction set architecture and start from the same arbitrary initial user-visible state.

The method uses a *projection function*, $Proj : (U \times P) \rightarrow U$, which removes all but the user-visible state from the implementation, and an *abstraction function*,

$$Abs(\langle \vec{u}, \vec{p} \rangle) \doteq Proj(Stall^l(\langle \vec{u}, \vec{p} \rangle)),$$

which maps the combined user-visible and pipeline state $\langle \vec{u}, \vec{p} \rangle$ of the implementation to its user-visible state. This is done by stalling the pipeline for a sufficient number of cycles l, that would allow all partially executed instructions to complete (so that the pipeline is flushed), and then stripping off all but the user-visible state. The correctness criterion expressed by Figure 1 is

$$\forall\ \vec{x},\ \vec{u},\ \vec{p}\ \ Abs(\delta_I(\vec{x}, \langle \vec{u}, \vec{p} \rangle)) = \delta_S(\vec{x}, Abs(\langle \vec{u}, \vec{p} \rangle)), \tag{1}$$

where $\vec{x}$ is an input combination that allows the implementation and the specification to execute one cycle without stalling, i.e., to start executing one instruction (and to complete it in the case of the specification).

3. Symbolic Domain

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 represented 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**:

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

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

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

The predicates *Hard* and *Soft* define the conditions for their arguments to be the ternary 1 and X, respectively. The predicate *Unique* defines the condition for the address set expression a to represent a *unique* or single address.

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

$$ITE(b,\ t,\ e) \doteq (b \wedge t) \vee (\neg b \wedge e)\,. \tag{5}$$

Address set comparison with another address set is implemented as:

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

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:

$$\begin{aligned} &a_1.h_i \leftarrow ITE(b,\ a_2.h_i,\ a_3.h_i)\,, \\ &a_1.l_i \leftarrow ITE(b,\ a_2.l_i,\ a_3.l_i)\,, \quad i = 1, \ldots, n\,. \end{aligned} \tag{7}$$

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

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

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

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

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

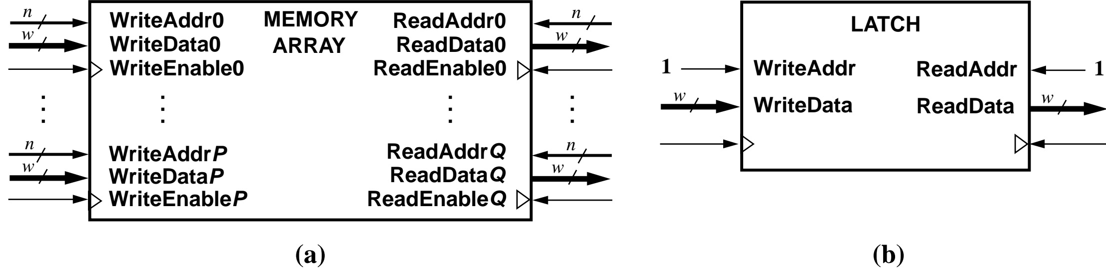

Figure 2. (a) A memory array that can be modeled by an EMM; (b) A latch modeled by an EMM

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* $d_1 \tilde{\cup} d_2$ of two data set expressions as:

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

We have used Ordered Binary Decision Diagrams (OBDDs) [4] to represent the Boolean expressions in our implementation. However, any representation of Boolean expressions can be substituted, as long as function *Valid*() can be defined for it.

4. Efficient Modeling of Memory Arrays in Symbolic Ternary Simulation

4.1 Overview

The EMM models memory arrays with write and/or read ports, all of which have the same numbers of address and data bits - n and w, respectively - as shown in Figure 2.a. An inward/outward triangle indicates an enable input of a write/read port. The assumption is that every memory system can be represented with such EMMs and possibly some extra logic. For example, a latch can be viewed as a memory array with a single address, so that it can be represented as an EMM with one write and one read port, both of which have the same number of data bits and only one address input, which is identically connected to the same constant logic value (e.g., **true**) - see Figure 2.b.

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 (i.e., **false**). In case of multiple port `Enables` not being 0 simultaneously, the resulting accesses to the memory array will be ordered according to the priority of the ports. It will be assumed that the memories respond instantaneously to any requests.

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 set expression denoting a memory location, and d is a data set expression representing the contents of this location. The context information is included for modeling memory systems where the *Write* and *Read* operations 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 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. These operations will be presented shortly.

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 $ITE(Hard(c),\ rd,\ ITE(Soft(c),\ (rd\ \tilde{\cup}\ d),\ d))$. Again, this guarantees that the EMM provides a conservative approximation of the replaced memory array. The routines needed by the software interface for accessing the list are presented next.

After completing a *Write* operation, the software interface checks every read port of the same memory for a possible on-going 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 $rd\ \tilde{\cup}\ d$ 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.

4.2 Memory Support Operations

The list entries are kept in order from *head* (low priority) to *tail* (high priority). Intuitively, the entries towards the low priority end correspond to the initial state of the memory, while the ones at the high priority end represent recent memory updates, with the tail entry being the result of the latest *Write* operation. Entries may be inserted at either end, using procedures *InsertHead*() and *InsertTail*().

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

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

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.

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

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

For each list entry, function *Read*() builds the Boolean expression *hard_match* 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.

$\mathcal{U}_{\mathcal{D}}$ 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 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 for a way to optimize the *Write* operation, the reader is referred to [19].

5. Comparing Symbolic Ternary Memory Execution Sequences

In Correspondence Checking, we wish to test whether two sequences of memory operations, which we will refer to as "A" and "B," yield identical behaviors. That is, we assume the two sequences start with identical initial memory states. For each *Read* operation in sequence A, its counterpart in sequence B must encounter the same initial value. Furthermore, the final states resulting from the two sequences must match. To implement this, we require some mechanism for guaranteeing that consistent values are used for the initial contents of the two memories. In addition, we require an algorithm for comparing the contents of two memories.

5.1 Maintaining Consistent Initial States

If we were to execute the operations for the two sequences independently, we would generate different symbols to represent the initial memory contents, and hence we would not yield matching results. Even if we could "reset" our symbol generator, so that the execution sequence B used the same series of generated symbols as sequence A, there would be a mismatch if the two sequences access memory locations in a different order. Instead, we modify the *Read* operation to maintain a consistent initial state between the memory being operated on, and a "shadow" memory, as shown in Figure 5.

function *ShadowRead*(**List** *mem*, **List** *shadow*,
ASExpr *a*) : **DSExpr**
/* Read from location *a* of memory *mem* */
/* Maintain consistency with shadow memory */
$g \leftarrow$ *GenDataExpr*(*mem*)
if $\neg$*Valid*($\neg$*Unique*(*a*)) **then**
InsertHead(*mem*, ⟨*Unique*(*a*), **false**, *a*, *g*⟩)
InsertHead(*shadow*, ⟨*Unique*(*a*), **false**, *a*, *g*⟩)
return *Read*(*mem*, *a*)

Figure 5. Implementation of the *ShadowRead* operation

Function *GenDataExpr*() is used to produce a data set expression consisting of a fresh Boolean variable for every bit position. This data set expression will be used as "initial" data for the contents of the given address under the contexts when the address is unique and it has not been initialized or written to before. In this way, we dynamically introduce state for symbolic memory locations, as needed by the circuit for a given simulation sequence.

Inserting the element ⟨*Unique*(*a*), **false**, *a*, *g*⟩ into the *head* (low priority) end of both lists ensures that subsequent *Read* operations will encounter the same data set expression for the "initial" state of location *a* as the one used by the present *Read*. As an optimization, when data set expression *g* has already been overwritten in both the primary and the shadow memory lists, it can be deleted from both lists and reused.

When executing the sequence A, we would use memory B as the shadow, and conversely when executing the sequence B, we would use memory A as the shadow. Note that the *Write* operations proceed as before. With this shadowing, any time a symbolic variable is assigned to represent the initial state of a memory location, the same symbol will be assigned to the same location and under the same contexts in both memories, thus enforcing the assumption that the two memories have matching initial states.

5.2 Comparing Final States in Symbolic Ternary Simulation

When adopting the correctness criterion (1) of Burch and Dill's method to symbolic ternary simulation, we can treat the final state Q'_{Spec}, reached along the specification side of the commutative diagram, as an overspecification of the final state Q''_{Spec}, reached along the implementation side of the diagram. In other words, the implementation will be correct if each of the state bits in Q''_{Spec} will be equal to or not more general than its counterpart bit in Q'_{Spec}, i.e., the two bits must be related according to the bit-level definition of the "$\sqsubseteq$"-relation from (8). Hence, the correctness criterion when combining Burch and Dill's method with ternary simulation can be expressed as:

$$\forall\ \vec{x},\ \vec{u},\ \vec{p}\ .\ \ Abs(\delta_I(\vec{x}, \langle\vec{u}, \vec{p}\rangle)) \sqsubseteq \delta_S(\vec{x}, Abs(\langle\vec{u}, \vec{p}\rangle))\,, \qquad (11)$$

which is equivalently represented in Figure 6.

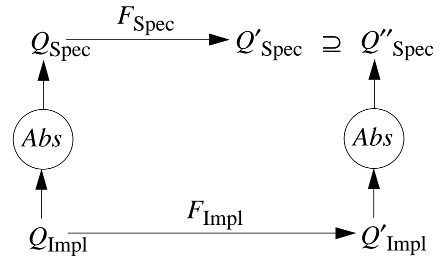

Figure 6. Commutative diagram for the correctness criterion when using symbolic ternary simulation

In comparing the contents of two memories, we can exploit the fact that only a small number of locations actually have defined values for any given context. Figure 7 shows how a Boolean expression can be constructed indicating the contexts for which two memories have corresponding contents. This algorithm only checks the locations denoted by the address set expressions occurring in the two lists, since all other addresses have not been accessed and their initial contents, assumed to be identical,

have not been modified. A further optimization, that is not shown, is to maintain a table of the address set expressions that have been processed, so as to ensure that only one comparison is performed for each distinct address set expression.

```
function CompareForContainment(List SpecMem,
                               List ImplMem) : BExpr
/* Compare implementation memory for */
/* correspondence to specification memory */
   correspondence ← true
   for each ⟨eh, es, ea, ed⟩ in SpecMem do
      Spec_d ← ShadowRead(SpecMem, ImplMem, ea)
      Impl_d ← ShadowRead(ImplMem, SpecMem, ea)
      correspondence ← correspondence ∧
                        (Impl_d ⊆ Spec_d)
   for each ⟨eh, es, ea, ed⟩ in ImplMem do
      Spec_d ← ShadowRead(SpecMem, ImplMem, ea)
      Impl_d ← ShadowRead(ImplMem, SpecMem, ea)
      correspondence ← correspondence ∧
                        (Impl_d ⊆ Spec_d)
   return correspondence
```

Figure 7. Comparing states of two memories in symbolic ternary simulation

5.3 Observation

One final subtlety about our comparison technique is worth noting. Normally two execution sequences will yield matching final memory states only if they perform identical *Write* operations, at least for the final writes to each memory location. Thus, if sequence A performs a write to some address a, one would expect sequence B to do likewise. Consider the case, however, where sequence A first reads from the unique address a and then writes the read value back to address a. Then location a is still in its initial state, and there is no need for sequence B to either read or write this location. Observe that our method will correctly handle this case. In executing sequence A, we will add entries ⟨*Unique*(a), **false**, a, g⟩ to both lists. The *Write* operation in A may cause this entry to be replaced, but since it preserves the initial state of this location, the two memories will compare successfully.

The condition described above is also the reason why the list for memory A must be used as a shadow argument for the *Read* operations performed on memory B. Even though we have already evaluated the effect of all *Read* and *Write* operations by sequence A, sequence B may access memory locations never accessed by A. This is allowed as long as the accesses do not alter the values at these or any other memory locations.

On the other hand, suppose sequence A writes to address a without ever reading the initial state, while sequence B never reads or writes this address. Then the list for A will contain an entry with address a, while the list for B will not. Executing *ShadowRead*(*memB*, *memA*, a) will return an expression involving a new data set expression generated by *GenDataExpr*(), which will not equal the expression returned by *ShadowRead*(*memA*, *memB*, a), and hence the mismatch will be detected. Notice that this situation will not result in an error when sequence A is produced by the specification circuit and the data that A writes to address a is $\mathcal{U}_{\mathcal{D}}$.

6. Generating Memory State

One drawback of OBDDs [4], used as a representation of Boolean expressions in our implementation of the presented algorithms, is that they require a global ordering of the variables. This translates into a problem when generating the initial state of EMMs, due to the widely varying classes of instructions supported by modern instruction set architectures.

The problem stems from an implicit rule that the address variables that will be used in an address set expression for accessing an EMM, need to be placed before the data variables that will represent the contents of that location. This requirement reduces the OBDD sizes, since the address variables will be used to select a set of data variables for the contents of the corresponding memory location, when function *Read*() is invoked. The selection will be based on the comparison with other address set expressions in the list for the EMM. If the address variables were intermixed with the data variables, or were situated after them in the global ordering of the variables, we would expect an exponential complexity of the Boolean expressions returned by function *Read*(), as opposed to a linear complexity. This would be due to the need for the data variables to be replicated in these expressions for every possible outcome of comparing the given address set expression to the address set expressions in the list.

A similar rule is that control variables (e.g., operation-code, and functional-code variables) need to be placed before the data variables that they will affect. Again, the control variables will select one out of many operations to be performed on the data variables, so that the above argument still applies.

In order to account for the above two rules, we introduce the notion of ordered variable groups. Each variable group contains interleaved vectors of symbolic Boolean variables that are used for the same purpose (e.g., addresses, control, data). Additional vectors from the same group can be generated dynamically, as required by the circuit for a given symbolic simulation sequence. The variable groups are ordered based on their relation, according to the above two rules.

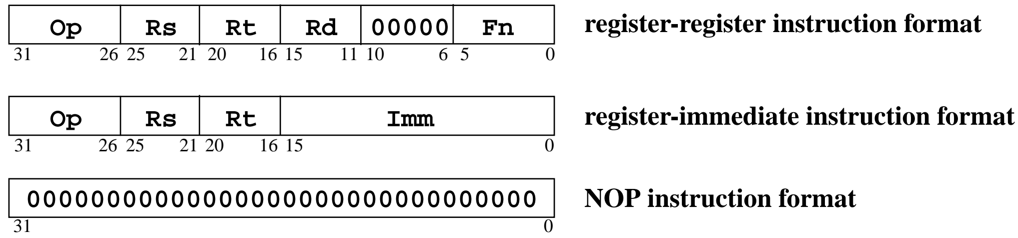

Figure 8. The MIPS formats for register-register, register-immediate, and NOP instructions. The instructions are encoded with 32 bits, numbered 0 through 31 - see the digits below the rectagles. Op stands for the operation-code field of 6 bits, Fn represents the functional-code field of 6 bits, and Imm is the immediate field of 16 bits. Rs, Rt and Rd are 5-bit register identifiers. Rs and Rt are the source registers, while Rd is the destination register, for register-register instructions. Rs is a source register providing the second operand, and Rt is the destination register for register-immediate instructions

Therefore, the ideal global ordering of the variable groups, needed to verify the MIPS pipeline [15] for its register-register and register-immediate instructions [10] (see Figure 8), will be: instruction-addresses, operation-codes, functional-codes, register-identifiers, high-data, and low-data (when listed from first to last in the global ordering of the variables). The need to split the data variables into two groups is dictated by the format of the MIPS register-immediate instructions. Notice that the instruction-address variables represent the contents of the program counter (PC) and identify locations of the instruction memory (IMem), so that they need to be placed before the instructions (that will represent the contents of the IMem), i.e., before all other variable groups. The justification for the ordering of the other variable groups is similar.

As Figure 8 shows, the lowest 6 bits of the MIPS instructions can represent the functional code in register-register instructions, or part of the immediate data in register-immediate instructions. Ideally, these 6 bits would be encoded with functional-code variables for the register-register instructions, and data variables for the register-immediate instructions. Furthermore, the functional-code variables need to be placed before the data variables (as the functional code determines what operations are to be performed on the data by the ALU). Supplying a single vector of symbolic Boolean variables for the initial state of the instruction memory at a particular address location (the way that function *GenDataExpr*() was defined to work so far) will mean that a dynamic-variable-reordering OBDD package will need to find a different global variable order, conditional on the type of the instruction, which will be impossible for a symbolic instruction representing many instruction formats simultaneously.

6.1 Variable-Group Indexing

Our solution is to introduce a state-encoding technique that we call "variable-group indexing". It allows the EMM to return a data set expression that consists of multiple data set expressions, each composed of variables from many variable groups and/or constants, which are indexed (selected) by the type of the instruction. The resulting data set expression is generated according to a pattern, defined by the user, for the given EMM. Figure 9 illustrates how this pattern can be defined for the IMem of the MIPS, when the set of verified instructions consists of `ori`, `or`, and `nop`.

```
gendataexpr IMem (
 Op  : OpCode
 Fn  : FnCode
 Rs, Rt, Rd : RegId
 Imm : LoData
 switch <Op, Fn> (
  case <0,0,1,1,0,1,-,-,-,-,-,->: /* ori */
    return <0,0,1,1,0,1, Rs, Rt, Imm>
  case <0,0,0,0,0,0,1,0,0,1,0,1>: /* or  */
    return <0,0,0,0,0,0, Rs, Rt, Rd,
          0,0,0,0,0,1,0,0,1,0,1>
  default:                        /* nop */
    return <0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0>
 )
 condition <Op, Fn> =
    <0,0,1,1,0,1,-,-,-,-,-,-> ||   /* ori */
    <0,0,0,0,0,0,1,0,0,1,0,1> ||   /* or  */
    <0,0,0,0,0,0,0,0,0,0,0,0>      /* nop */
)
```

Figure 9. Definition of the variable-group indexing pattern for the MIPS instruction memory (IMem), when verifying the pipeline for the instructions ori, or, and nop. "-" indicates a don't-care, so that the corresponding variable does not participate in the Boolean expression

`OpCode`, `FnCode`, `RegId`, and `LoData` are the declaration names for the operation-code, functional-code, register-identifier, and low-data variable groups, respectively. `Op`, `Fn`, `Rs`, `Rt`, `Rd`, and `Imm` are vectors of fresh Boolean variables generated within the corresponding

variable group by being interleaved with other such vectors in the same variable group. The `switch` statement uses the concatenation of the `Op` and `Fn` vectors of Boolean variables in order to index one out of three possible patterns for the initial state of the IMem, according to the `case` and `default` directives. The `condition` statement defines a Boolean expression in terms of the fresh variables in `Op` and `Fn`, which is used to update a Boolean expression *global_care_set* (whose application will be illustrated shortly) by being conjuncted with the current value of *global_care_set* every time when a new data set expression is generated for that memory. In this way, the `switch` and `condition` statements allow the state of the EMMs to be restricted to a subset of the system state space. This makes possible the verification of microprocessors for a subset of their instruction set architectures.

Hence, function *GenDataExpr*() is modified in order to reflect the variable-group indexing pattern for an EMM, i.e., to return a Boolean expression instead of just a single Boolean variable for every bit position in the generated data set expression.

As Figure 9 shows, we might be imposing restrictions on the initial state that is generated on-the-fly by *GenDataExpr*(), therefore considering a subset of the system state space as a possible initial state. However, the correctness criterion expressed by the commutative diagram from Figure 1 is a proof by induction. It proves that if the implementation starts from an arbitrary initial state Q_{Impl}, and is exercised with an arbitrary instruction, then the reachable state Q'_{Impl} will be correct, when compared (by means of an abstraction function) to the specification's behavior for an arbitrary instruction. Since the initial state Q_{Impl} is arbitrary, then it is a superset of the reachable state Q'_{Impl}, so that, by induction, the implementation will function correctly for an arbitrary instruction applied to Q'_{Impl}. However, if the initial state Q_{Impl} is restricted to a subset of the system state space, and Q'_{Impl} is not a subset of Q_{Impl}, then the inductive argument for the correctness criterion would not hold. Hence, we need to check that Q'_{Impl} is a subset of Q_{Impl}. This is equivalent to proving an invariant of the implementation.

6.2 Verifying Memory State for Consistency with Its Variable-Group Indexing Pattern

We therefore modify the *Write* operation in order to check, conditional on a new argument Boolean expression *verify_generality*, the set of reachable states for being a subset of the initial state, as expressed by the variable-group indexing pattern for the corresponding memory - see Figure 10. The type **Boolean** is used for flags that can be either **true** or **false**.

procedure *Write*(**List** *mem*, **CExpr** *c*, **ASExpr** *a*, **DSExpr** *d*,
  BExpr *global_care_set,* **Boolean** *verify_generality*)
/* Write data *d* to location *a* under control *c* */
  $h \leftarrow Hard(c) \wedge Unique(a)$
  $s \leftarrow Soft(c) \;\vee\; Hard(c) \wedge \neg Unique(a)$
  if (*verify_generality*) **then**
    VerifyGenerality(*mem*, *h*, *s*, *d*, *global_care_set*)
  InsertTail(*mem*, ⟨*h, s, a, d*⟩)

Figure 10. Modified implementation of the *Write* operation

Procedure *VerifyGenerality*(), shown in Figure 11, verifies its data set expression argument for being a subset (according to the "⊆"-relation from (8)) of the initial state for the corresponding memory, as represented by a fresh data set expression *g*, generated according to the variable-group indexing pattern for the memory. Function *GetVariableSupport*() extracts the symbolic Boolean variables used in its argument data set expression. For the new data set expression *d* to be a subset of the initial state of the memory, we require that for every binary pattern of the variables in the support of *d*, there exist a binary pattern of the variables in the support of the newly generated default data set expression *g* that makes *g* be a superset of *d*, given the implication by the Boolean expression *global_care_set*, conjuncted with $(h \vee s)$. Notice that the universal quantification with respect to the variables in the support of *d* is done implicitly. Given that the result of the existential quantification with respect to the variables of *g* is not **true**, then a further universal quantification would never make it **true**. The verification is aborted by calling procedure *Exit*(), when the Boolean expression *correct* is not **true**, i.e., when the data set expression *d* is not a subset of the initial state for the memory. As an optimization, the data set expression *g*, generated in the procedure, can be reused.

procedure *VerifyGenerality*(**List** *mem,* **BExpr** *h,* **BExpr** *s,*
  DSExpr *d,* **BExpr** *global_care_set*)
/* Compare new memory state *d* for satisfying the memory's variable-group indexing pattern */
  $g \leftarrow GenDataExpr(mem)$
  $\vec{l} \leftarrow GetVariableSupport(g)$
  $correct \leftarrow \exists \vec{l}\,.\,[global_care_set \wedge (h \vee s) \;\Rightarrow\; (d \subseteq g)]$
  if $\neg$*Valid*(*correct*) **then**
    Exit()

Figure 11. Verifying new state of a memory for being a subset of the initial state of that memory, as represented by the memory's variable-group indexing pattern

Observe that procedure *VerifyGenerality*() need only be called for EMMs that get written to by the circuit and whose variable-group indexing patterns exclude some

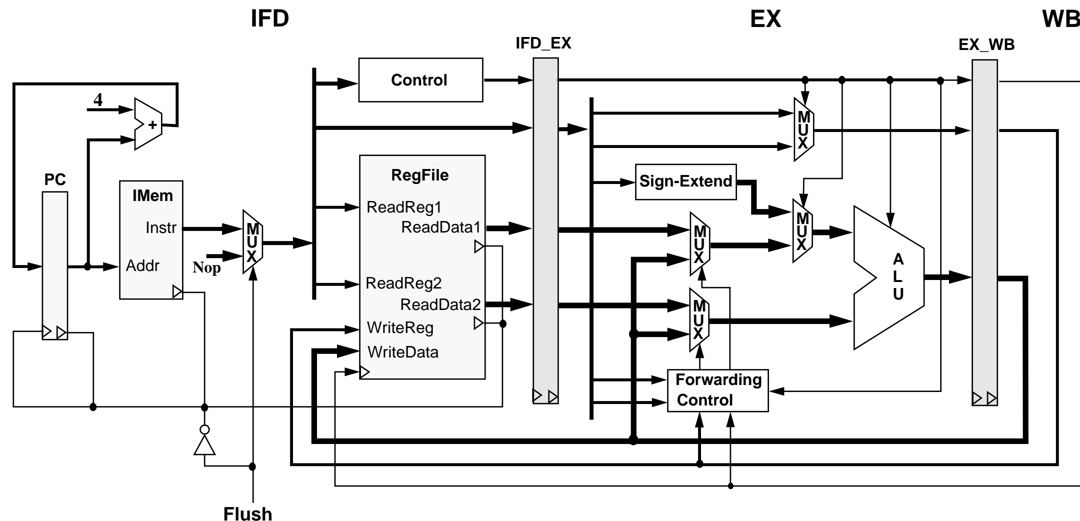

Figure 12. The shortened pipelined MIPS (SPMIPS)

combinations. Hence, these are only the pipeline latches in the MIPS.

7. Ternary Correspondence Checking by Applying Memory Shadowing

When applying memory shadowing, the EMM software interface uses function *ShadowRead*() for performing reads, and procedure *Write*() for performing writes. *ShadowRead*() provides the two execution sequences with identical initial memory state by constructing it on-the-fly. We check the correctness criterion (11) by applying function *CompareForContainment*() on all the user-visible memory elements. The universal quantification is done implicitly by using the same symbolic initial memory state and the same symbolic instruction for both execution sequences.

The steps of our methodology are:

1. Load the implementation (possibly pipelined) circuit and associate every memory element in it with empty original and shadow memory lists. $global_care_set \leftarrow$ **true**.
2. $verify_generality \leftarrow$ **true**. Simulate the implementation circuit for one clock cycle with a (legal) symbolic instruction. $verify_generality \leftarrow$ **false**.
3. Simulate a flush of the implementation circuit.
4. Swap the original and shadow memory lists for every memory element.
5. Simulate a flush of the implementation circuit.
6. Swap the implementation and the specification (non-pipelined) circuits by keeping the contents of the memory lists for every user-visible memory element.
7. Simulate the specification circuit for one clock cycle with the same symbolic instruction as used in Step 2. This would be provided automatically by function *ShadowRead*().
8. Compare the original, mem_i, and the shadow, $shadow_i$, memory lists for every user-visible memory element i and let $containment_i \leftarrow$ $$CompareForContainment(mem_i, shadow_i),$$ $i = 1, ..., u$, where u is the number of user-visible memory elements.
9. Form the Boolean expression for our correctness criterion: $$global_care_set \Rightarrow \bigwedge_{i=1}^{u} containment_i, \qquad (12)$$ where *global_care_set* is updated by function *GenDataExpr*() according to the `condition` statements in the declaration of variable-group indexing patterns for EMMs.

It is also possible to traverse the commutative diagram with another sequence of circuit and memory swaps, i.e., to exercise first the specification side of the diagram.

8. Experimental Results

We implemented all the correspondence checking routines, presented in this paper, within a tool [9] that supports the STE technique. Although correspondence checking and STE are two different forms of verification, as noted in Section 1, they have in common the use of a symbolic simulator and the EMM. This allows them to be applied on the same circuit descriptions, which can be in either gate-level or register-transfer-level form. Furthermore, gate-level circuits can be automatically generated from transistor-level circuits [3].

We examined the shortened pipelined MIPS (SPMIPS), shown in Figure 12. It was compared to its

Instructions Verified										CPU Time (s)					Memory (MB)				
										Data Path Width					Data Path Width				
or	and	add	slt	sub	ori	andi	addi	slti	nop	4	8	16	32	64	4	8	16	32	64
									✔	3	4	7	12	23	1.4	1.4	1.4	1.5	1.6
					✔				✔	6	11	20	42	91	2.1	2.8	4.4	7.4	13.5
✔									✔	12	23	48	102	224	4.3	7.3	13.3	25.4	49.4
✔					✔				✔	21	45	98	213	509	7.3	13.3	25.3	49.4	97.4
✔	✔				✔				✔	37	80	171	456	1 079	9.3	21.4	49.3	97.3	193.2
✔	✔				✔	✔			✔	45	92	203	480	1 256	13.3	25.3	49.3	97.3	193.4
✔	✔	✔			✔	✔			✔	68	148	320	829	---	17.1	34.0	57.3	145.6	---
✔	✔	✔			✔	✔	✔		✔	79	186	403	1 056	---	13.3	40.3	87.8	150.1	---
✔	✔	✔		✔	✔	✔	✔		✔	113	268	659	---	---	25.3	49.3	113.2	---	---
✔	✔	✔	✔	✔	✔	✔	✔		✔	141	299	789	---	---	29.3	49.3	133.8	---	---
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	164	419	1 141	---	---	41.3	97.1	193.2	---	---

Table 2. Experimental results from the correspondence checking between the SPMIPS and its non-pipelined version when verifying the instructions marked with "✔". In the cases indicated with "---", the experiments did not finish because of limited/insufficient memory

non-pipelined version. The shaded boxes represent memory arrays and latches modeled by EMMs. The SPMIPS consists of three stages: a combined instruction fetch and decode stage (IFD), an execution stage (EX), and a write-back stage (WB). The original MIPS is described in [15].

Experiments for correspondence checking of the SPMIPS to its non-pipelined version were performed for data path widths (i.e., the data width of the register file and the ALU) between 4 and 64 bits. The instruction size was always 32 bits, as in the original MIPS. Two MIPS instruction formats were supported by the SPMIPS and its non-pipelined version - register-register and register-immediate. Particularly, ten instructions were implemented in the control of both microprocessors: five register-register instructions - `or`, `and`, `add`, `slt`, and `sub`; four register-immediate instructions - `ori`, `andi`, `addi`, and `slti`; and the `nop`. The immediate operand in register-immediate instructions was formed from the 16-bit immediate field of the instruction by taking as many bits as half the data path width, but not exceeding 16 (in the case of a 64-bit wide data path), and sign-extending them by replicating the most significant bit until filling the data width.

The multiplexor controlled by signal `Flush` is assumed to have been added to the circuit by the user in order to implement the flushing function. Furthermore, we modified the circuit by conjuncting the complement of signal `Flush` with some EMM port enable signals (this is implicitly shown as connecting the complement of `Flush` to the corresponding port enables). In this way, we did not read/write the PC and did not read the IMem and the RegFile during flushing. This is allowed, since the method requires an arbitrary abstraction function that makes the diagram for the correctness criterion commute. However, the above flushing function reduces the CPU time and memory requirements for the verification. Observe that the PC should be read and written during flushing if jump or brunch instructions are also being verified.

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. Table 2 shows the results from the correspondence checking between the SPMIPS and its non-pipelined version. Some of the experiments did not finish because of a system limitation that the C function `malloc()` cannot allocate more than 256 MB of memory for a single process, although the computer that we used had 512 MB of physical memory.

9. Conclusions and Future Work

Based on our experiments, we can draw the following conclusions:

- Register-immediate instructions take less CPU time and memory to be verified than do register-register

instructions. We explain that with their simpler variable-group indexing patterns.

- The CPU time and memory approximately double as the data path width doubles.
- The CPU time and memory flatten as the set of verified instructions is expanded. The reason is the simplification of the Boolean expressions when generating the initial state of the EMMs because of the availability of more instructions with the same format, but slightly different operation-codes and/or functional-codes.
- An extra level of forwarding logic, due to a dummy pipeline stage between the EX and WB stages of the SPMIPS, increased the CPU time and memory from two to more than ten times for the experiments that could finish (not shown). This can be attributed to the greater complexity of the Boolean expressions for the resulting ALU operands.

In the future, we will work on automating the process of defining the variable-group indexing patterns for pipeline latches, in order to reduce the chance for an error on the side of the user. We will also study the efficiency of functions that simplify OBDDs based on a care-set OBDD. Furthermore, we will explore ways to deal with the verification complexity introduced by multiple levels of forwarding logic. Finally, we will work on techniques to enable the verification of load, store, jump, and branch instructions. Notice that the variable-group indexing patterns of these instructions cannot satisfy the rule for placing address variables before data variables in the global variable ordering. The reason is that variables which will represent the data of a memory array in the variable-group indexing patterns of these instructions, will also be used as an address to that same memory array. For example, in the case of the store instruction, the contents of a register will serve both as data to be stored in the data memory and as a destination address in the same memory.

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] S. Bose, and A.L. Fisher, "Verifying Pipelined Hardware Using Symbolic Logic Simulation," *International Conference on Computer Design*, October 1989, pp. 217-221.

[3] R.E. Bryant, "Extraction of Gate Level Models from Transistor Circuits by Four-Valued Symbolic Analysis," *International Conference on Computer Aided Design*, November 1991, pp. 350-353.

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

[5] R.E. Bryant, and M.N. Velev, "Verification of Pipelined Microprocessors by Comparing Memory Execution Sequences in Symbolic Simulation,"[2] *Asian Computer Science Conference* (*ASIAN '97*), R.K. Shyamasundar and K. Ueda, eds., LNCS 1345, Springer-Verlag, December 1997, pp. 18-31.

[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] J.R. Burch, "Techniques for Verifying Superscalar Microprocessors," *DAC '96*, June 1996, pp. 552-557.

[8] C.A.R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica*, 1972, Vol.1, pp. 271-281.

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

[10] G. Kane, and J. Heinrich, *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[11] K.L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[12] R.B. Jones, C.-J.H. Seger, and D.L. Dill, "Self-Consistency Checking," *FMCAD '96*, M. Srivas and A. Camilleri, eds., LNCS 1166, Springer-Verlag, November 1996, pp. 159-171.

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

[14] 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.

[15] D.A. Patterson, and J.L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2nd Edition, Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[16] J. Sawada, and W.A. Hunt, Jr., "Trace Table Based Approach for Pipelined Microprocessor Verification," *CAV '97*, O. Grumberg, ed., LNCS 1254, Springer-Verlag, June 1997, pp. 364-375.

[17] 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.

[18] 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.

[19] M.N. Velev, and R.E. Bryant, "Efficient Modeling of Memory Arrays in Symbolic Ternary Simulation,"[2] *First International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, Portugal, March-April 1998.

[20] M.N. Velev, and R.E. Bryant, "Efficient Modeling of Memory Arrays with Timing Requirements in Symbolic Ternary Simulation,"[2] *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU '97)*, Austin, TX, December 1997, pp. 28-38.

[21] P.J. Windley, and J.R. Burch, "Mechanically Checking a Lemma Used in an Automatic Verification Tool," *FMCAD '96*, M. Srivas and A. Camilleri, eds., LNCS 1166, Springer-Verlag, November 1996, pp. 362-376.

2. Available from: `http://www.ece.cmu.edu/afs/ece/usr/mvelev/.home-page.html`