Appears in the proceedings of *Asian Computer Science Conference* (*ASIAN '97*), R.K. Shyamasundar and K. Ueda, eds., LNCS 1345, Springer-Verlag, December 1997, pp. 18-31.

Verification of Pipelined Microprocessors by Comparing Memory Execution Sequences in Symbolic Simulation[1]

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

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

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

Abstract. This paper extends Burch and Dill's pipeline verification method [4] to the bit level. We introduce the idea of memory shadowing, a new 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 equality. Memory shadowing and the comparison algorithm build on the Efficient Memory Model (EMM) [13], a behavioral memory model where the number of symbolic variables used to characterize the initial state of a memory is proportional to the number of distinct symbolic locations accessed. These techniques allow us to verify that a pipelined circuit has equivalent behavior to its unpipelined specification by simulating two memory execution sequences and comparing their final states. Experimental results show the potential of the new ideas.

Keywords: pipelined microprocessor verification, memory shadowing, Efficient Memory Model (EMM), circuit correspondence checking, symbolic simulation.

1. Introduction

We are extending Burch and Dill's pipeline verification method [4] to a bit-level circuit verification. The idea of a commutative diagram and an underlying abstraction function, used by Burch and Dill, is not new to verification. It has been introduced by Hoare [6] for verifying computations on abstract data types in software, and has been used by Bose and Fisher [1] 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* (see Figure 1). 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.

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,

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

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}.

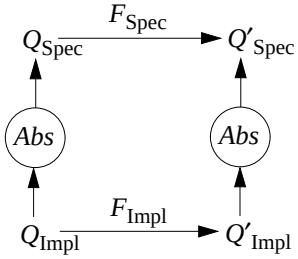

Figure 1. Commutative diagram for the correctness criterion.

Burch and Dill's implementation [4][5] 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. A naive implementation of Burch and Dill's method at the bit-level would require introducing a symbolic Boolean variable for every bit of register or memory state. This would lead to unacceptable complexity. Our paper overcomes this limitation by using the Efficient Memory Model (EMM) [13] to represent memory state.

The EMM is a behavioral memory model where the number of symbolic variables used to characterize the initial state of a memory is proportional to the number of distinct symbolic locations accessed, rather than to the size of the memory. It is based on the observation that a single execution sequence used in formal verification typically 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 our symbolic simulator.

Burch and Dill also use a symbolic representation of memory arrays in their implementation [4]. 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.

An extensive body of research has been spawned by Burch and Dill's method. Sawada and Hunt [11] 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 [5] has extended it to superscalar processor verification by proposing a new flushing mechanism and by decomposing the commutative diagram from [4] into three more easily verifiable commutative diagrams. The correctness of this decomposition is proven by Windley and Burch [14]. Jones, Seger, and Dill [8] 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) the memory shadowing technique for ensuring identical initial memory state encountered by two different memory execution sequences; 2) an algorithm to compare the final states of two memories; 3) a methodology that extends Burch and Dill's pipeline verification method [4] to efficiently model the complete functionality of the data path at the bit level; and 4) experimental results that confirm the applicability of the new ideas.

We consider two forms of verification: 1) *Symbolic Trajectory Evaluation* (STE) [12], 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 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 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. Our previous paper describes the use of the EMM in this context [13]. 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.

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 EMM. The memory shadowing technique for providing on-the-fly identical initial memory state to

two different execution sequences is explained in Section 5, which also presents an algorithm that compares for equality the states of two memory arrays. The verification methodology for correspondence checking by applying memory shadowing is described in Section 6. Experimental results are presented in Section 7. Finally, conclusions are made and future work is outlined in Section 8.

2. Burch and Dill's Pipeline Verification Method

For the purpose of verifying a pipelined processor, Burch and Dill [4] 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 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 as many cycles as its depth *l*, so that it can be flushed, and then stripping off all but the user-visible state. The correctness criterion expressed by Figure 1 is

$$\forall\ \vec{x},\ \vec{u},\ \vec{p}\ .\ \ Abs(\delta_I(\vec{x}, \langle \vec{u}, \vec{p} \rangle)) \stackrel{?}{=} \delta_S(\vec{x}, Abs(\langle \vec{u}, \vec{p} \rangle)), \qquad (1)$$

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

expressions in the algorithms to be presented.

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

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

$$ITE(b, t, e) \doteq (b \wedge t) \vee (\neg b \wedge e). \quad (2)$$

Address comparison is then implemented as:

$$A1 = A2 \doteq \neg \bigvee_{i=1}^{n} A1_i \oplus A2_i, \quad (3)$$

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

$$A1_i \leftarrow ITE_i(b, A2, A3) \doteq A1_i \leftarrow (b \wedge A2_i) \vee (\neg b \wedge A3_i), \quad i = 1, ..., n. \quad (4)$$

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

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

4. Efficient Modeling of Memory Arrays

4.1 Overview

The assumption of the EMM [13] 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 (see Figure 2).

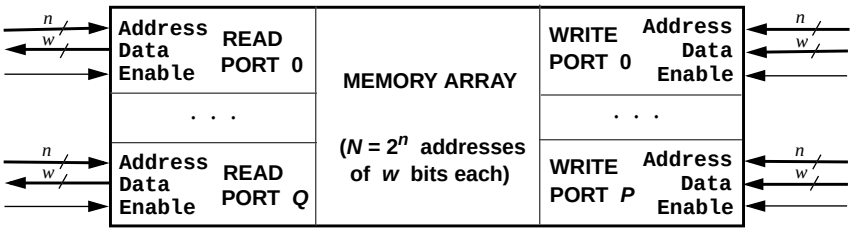

Figure 2. View of a memory array as an EMM.

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 port 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**.

The interaction of the memory array with the rest of the circuit is assumed to take place on the rising edge of a port `Enable` signal. In case of multiple port `Enable`s

having rising edges simultaneously, the resulting accesses to the memory array will be ordered according to the priority of the ports.

During symbolic simulation, the memory state is represented by a list containing entries of the form $\langle c, a, d \rangle$, where c is a Boolean expression denoting the set of contexts for which the entry is defined, a is an address expression denoting a memory location, and d is a data expression denoting the contents of this location. The context information is included for modeling memory systems where the *Write* and *Read* operations may be performed conditionally depending on the value of a control signal. Initially the list is empty.

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

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 memory *Write* operation. Entries may be inserted at either end, using procedures *InsertHead* and *InsertTail*, and may be deleted using procedure *Delete*.

The function *Valid*, when applied to a Boolean expression, returns **true** if the expression is valid, i.e. true for all contexts, and **false** otherwise. Note that in all of the algorithms, a Boolean expression cannot be used as a control decision in the code, since it will have a symbolic representation. On the other hand, we can make control decisions based on whether or not an expression is valid.

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

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

The *Write* operation, shown as a procedure in Figure 3, takes as arguments a memory

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

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

Figure 3. Implementation of the *Write* operation.

The *Read* operation is shown in Figure 4 as a function which, given a memory list, a Boolean expression denoting the contexts for which the read should be performed, and an address expression, returns a data expression indicating the contents of this location.

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

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

Figure 4. Implementation of the *Read* operation.

The main part of the *Read* operation is implemented with the function *ReadWithDefault*. The purpose of *ReadWithDefault* is to construct a data expression giving the contents of the memory location denoted by its argument address expression. It does this by scanning through the list from lowest to highest priority, adding a selection operator to the expression that chooses between the list element's data expression and the previously formed data expression, based on the match condition. It also generates a Boolean expression *found* indicating the contexts for which a matching list element has been encountered. *ReadWithDefault* has as its fourth argument a "default" data expression to be used when no matching list element is found. When this case arises, a new list element is inserted into the *head* (low priority) end of the list.

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

5. Comparing Memory Execution Sequences

In some applications, 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 matching initial memory states. For each externally visible *Read* operation in sequence A, its counterpart in sequence B must return the identical 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.

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 context in both

memories, thus enforcing the assumption that the two memories have matching initial states.

```
function ShadowRead(List mem, List shadow, BExpr c, AExpr a): DExpr
/* Read from location a under context c */
/* Maintain consistency with shadow memory */
    g ← GenDataExpr()
    ReadWithDefault(shadow, c, a, g)
    return ReadWithDefault(mem, c, a, g)
```

Figure 5. Implementation of the *Read* operation when initial state consistency between two memories must be maintained.

5.2 Comparing Final States

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 6 shows function *CompareMem* which constructs a Boolean expression indicating the contexts for which two memories have matching contents. This code only checks the locations denoted by the set of address expressions occurring in the two lists. As a further optimization, it maintains a table *tested* to ensure that only one comparison is performed for each unique address expression.

```
function CompareMem(List MemA, List MemB): BExpr
/* Compare states of two memories */
    same ← true
    tested ← ∅
    for each ⟨ec, ea, ed⟩ in MemA do
    if (ea ∉ tested) then
        g ← GenDataExpr()
        da ← ReadWithDefault(MemA, ec, ea, g)
        db ← ReadWithDefault(MemB, ec, ea, g)
        same ← same ∧ (da =db)
    tested ← tested ∪ {ea}
    for each ⟨ec, ea, ed⟩ in MemB do
    if (ea ∉ tested) then
        g ← GenDataExpr()
        da ← ReadWithDefault(MemA, ec, ea, g)
        db ← ReadWithDefault(MemB, ec, ea, g)
        same ← same ∧ (da =db)
    tested ← tested ∪ {ea}
    return same
```

Figure 6. Comparing states of two memories.

CompareMem compares matching locations in the two memories using the function *ReadWithDefault*, with a newly generated symbolic data expression *g* as the default value. This operation will add a list element and return a data expression dependent on *g* only when either some *Write* operation has been performed with context argument $c \neq$ **true**, or some *Write* operation was performed to one memory, without a counterpart for the other. By using the same, newly-generated symbol for a pair of accesses, we maintain consistency between the initial states of the two memories as well as the property that each memory location can have an arbitrary initial value.

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 address *a* and then writes that 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 wil correctly handle this case. In executing sequence A, we will add entries ⟨**true**, *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 *ReadWithDefault*(*memB*, *a*, *g*) will return an expression involving *g*, which will not equal the expression returned by *ReadWithDefault*(*memA*, *a*, *g*), and hence the mismatch will be detected.

6. 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 (1) of Burch and Dill's method by applying function *CompareMem* 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.

2. Cycle the implementation with a symbolic instruction.
3. Flush the implementation.
4. Swap the original and shadow memory lists for every memory element.
5. Flush the implementation.
6. Swap the implementation and the specification (non-pipelined) circuits by keeping the memory lists for every user-visible memory element.
7. Cycle the specification with the same symbolic instruction as used in Step 2.
8. Compare the original, mem_i, and the shadow, $shadow_i$, memory lists for every user-visible memory element i and let $equality_i \doteq CompareMem(mem_i, shadow_i)$, $i = 1, ..., u$, where u is the number of user-visible memory elements.
9. Form the Boolean expression for our correctness criterion:

$$legal\_instruction \Rightarrow \bigwedge_{i=1}^{u} equality_i, \tag{5}$$

where *legal_instruction* is a Boolean expression for the symbolic instruction from Steps 2 and 7 to be legal.

It is also possible to traverse the commutative diagram with another sequence of circuit and memory swaps, i.e. to first flush the implementation, swap it with the specification, cycle the specification, swap it with the implementation, swap the memory lists, and perform Steps 2, 3, 8, and 9 from above. As expected, our experiments found that the two sequences of traversing the commutative diagram perform comparably in terms of CPU time and memory for our simple circuit presented next.

7. Experimental Results

We implemented all the correspondence checking routines, presented in this paper, within a tool [7] 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 both gate-level and register-transfer-level form. Furthermore, gate-level circuits can be automatically generated from transistor-level circuits [2].

Experiments were performed on the pipelined addressable accumulator shown in Figure 7. The current instruction is specified by the inputs `Addr`, `Clear`, `In`, and `Nop`, where the last one indicates whether the instruction is a nop and is used for flushing the pipeline. The pipeline register `Hold` separates the execution and the write back stages of the processor. The control logic stores the previous address and compares it with the present one at the `Addr` input. In case of equality of the two addresses and a valid previous instruction (the Nop input was **false**), the control signal of the multiplexor is set so as to select the data output of the `Hold` register. Hence, data forwarding takes effect. For a more detailed description of the circuit (however without a Nop

input) and its verification by STE, the reader is referred to [7] for the case of transistor-level memory elements, and to [13] for the case of EMM-replaced memory elements.

For all of the experiments, the dual-ported register file was removed from the circuit and replaced with an EMM. The software interface ensures that: 1) a *Read* operation takes place relative to `phi1`; and 2) a *Write* operation takes place relative to `phi2`, as long as the corresponding instruction was not a Nop - see the register file connections shown in Figure 7.(b).

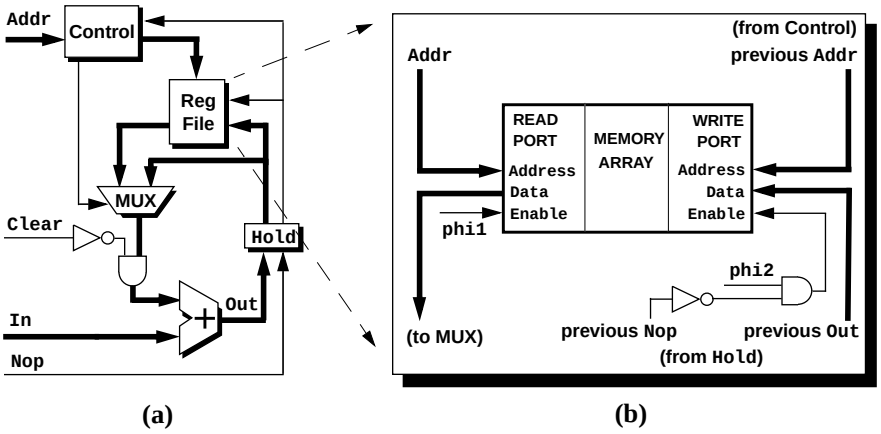

Figure 7. (a) The pipelined addressable accumulator; (b) the connections of its register file when replaced by an EMM. The thick lines indicate buses, while the thin ones are of a single bit.

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 1 shows the results from the STE verification of the pipelined addressable accumulator. Table 2 - from the correspondence checking between the same circuit and its non-pipelined version (the specification circuit) by applying memory shadowing. Finally, Table 3 presents the results from the STE verification of the specification circuit. In all of the tables, *N* is the number of addresses and *w* is the number of data bits per address.

It can be observed that the results in Table 1 depend on *N*, while those in Tables 2 and 3 are almost constant with *N*. The reason is that for the experiments for Table 1, the `RegFile` and the `Hold` latch are initialized conditionally on the equality of the current and previous addresses, as opposed to unconditionally which is the case in the experiments for Tables 2 and 3. The idea is that these conditions will cancel the effect of the forwarding logic, and the output of the multiplexor will be simple (see [7] and [13] for details). However, when the `RegFile` and the `Hold` latch are read, the initialization conditions get conjuncted with the contents of every data bit. Hence, the BDDs get bigger, require more CPU time to process, and the results depend on *N*.

N	CPU Time (s)				Memory (MB)			
	w				w			
	16	32	64	128	16	32	64	128
16	33	65	129	259	1.8	2.4	3.5	5.8
32	38	75	147	300	2.2	2.4	3.5	5.8
64	51	99	200	402	2.2	3.2	5.1	9.0
128	83	163	324	661	2.3	3.3	5.2	9.2

Table 1. Experimental results for the pipelined addressable accumulator, verified by STE.

N	CPU Time (s)				Memory (MB)			
	w				w			
	16	32	64	128	16	32	64	128
16	19	38	75	151	1.7	2.1	3.0	4.8
32	20	38	76	152	1.8	2.4	3.6	6.0
64	20	39	77	153	1.8	2.4	3.7	6.1
128	20	39	77	153	1.8	2.4	3.7	6.1

Table 2. Experimental results for the pipelined addressable accumulator, verified by correspondence checking with its non-pipelined version.

N	CPU Time (s)				Memory (MB)			
	w				w			
	16	32	64	128	16	32	64	128
16	23	46	91	183	1.5	1.7	2.1	3.0
32	23	46	92	183	1.5	1.7	2.2	3.0
64	23	46	91	183	1.5	1.8	2.3	3.3
128	24	47	92	186	1.5	1.8	2.3	3.4

Table 3. Experimental results for the non-pipelined addressable accumulator, verified by STE.

8. Conclusions and Future Work

We are very encouraged by our results. Correspondence checking between the pipelined addressable accumulator and its non-pipelined version required CPU time and memory that are logarithmic with respect to N, and linear with respect to w. By comparing the sum of the same entries in Tables 2 and 3 to their corresponding entry in Table 1, it can be concluded that for pipelined processors with sufficiently large memory state, it may take less CPU time and memory to verify an equivalent non-pipelined circuit and then to check it for correspondence to the pipelined one, than to directly

verify the pipelined processor. Furthermore, when the pipelined processor is incrementally modified, it can directly be checked for correspondence to its non-pipelined version, assuming the latter is already verified, and the savings in CPU time and memory will be even greater.

Future work may focus on applying the memory shadowing methodology on real-life processors. Crucial for that will be techniques for resolving the conflicting orderings of variables generated by function *GenDataExpr*, when representing the initial state of the pipeline registers. Namely, some of the instruction bits may correspond to both the functional code in one class of instructions and to a part of an immediate data operand in another class. The variables generated in the former case, since used in the control of the processor, will require to be towards the front of the variable ordering. However, the ones generated in the latter case, since used in the data path, will be more efficiently placed around the end of the variable ordering.

References

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

[2] 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.

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

[4] 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.

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

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

[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] 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.

[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] 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.

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

[14] 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.