A COMPARATIVE APPROACH TO PROCESSOR VERIFICATION USING SYMBOLIC MODEL CHECKING

by

Nagisa Ishiura and Robert K. Brayton

Memorandum No. UCB/ERL M94/59

24 August 1994

A COMPARATIVE APPROACH TO PROCESSOR VERIFICATION USING SYMBOLIC MODEL CHECKING

by

Nagisa Ishiura and Robert K. Brayton

Memorandum No. UCB/ERL M94/59

24 August 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A COMPARATIVE APPROACH TO PROCESSOR VERIFICATION USING SYMBOLIC MODEL CHECKING

by

Nagisa Ishiura and Robert K. Brayton

Memorandum No. UCB/ERL M94/59

24 August 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A Comparative Approach to Processor Verification Using Symbolic Model Checking

Nagisa Ishiura and Robert K. Brayton

Department of EECS, University of California at Berkeley, CA 94720

We present a method for verifying the correctness of processor designs using symbolic model checking. The correctness of register transfer level implementations of pipelined processors is verified by comparing their output signal sequences with those of corresponding unpipelined reference processors. For this purpose, several specification-implementation relations that formalize equivalences between signal sequences are defined taking time lags and irrelevant data into account. The algorithms to check the equivalence relations are given in the form of finite state machines called matching modules. A model is constructed consisting of the processor under verification, a reference processor, and the checker consisting of several matching modules. Verification is achieved by proving the output "ok" of the checker is always 1 for every possible program and data.

It is infeasible to handle the model consisting of processors including entire data paths. In order to solve this problem, several methods for simplifying (abstracting) data paths and memories are proposed and analyzed for accuracy. Preliminary experiments were conducted on a subset of the DLX processor using the CTL model checker *SMV*. Design errors associated with an update conflict of the program counter, cancellation of prefetched instructions on branch and jump, and pipeline control to avoid data hazard were detected using from 2 to 15 hours of CPU time with up to 40 MB of memory on a DECstation 5000/260 with 480MB.

1 Introduction

In order to develop competitive high-performance microprocessors, a variety of sophisticated architectures and implementation techniques [HP90] have been contrived. This, on the other hand, makes it more difficult to check design errors simply by logic simulation, which can cover only a very small portion of the entire space of possible instruction sequences and memory configurations. In this paper, we address a method for automatically verifying the behavior of processors with pipelined architectures.

There are basically two categories of processor verification; those using theorem provers [Cyr93] and those based on Boolean function manipulation and FSM (finite state machine) traversal [Ael92, Bea94, Bha94]. The utilization of abundant data types and human intuition in the theorem prover approach is attractive but much expert labor is required. On the other hand, the latter approach allows almost automatic verification. With recent progress in Boolean function manipulation [Bry86, Rud93] and implicit state traversal techniques [Tou90, Mcm93] using BDDs (binary decision diagrams), this approach has become powerful enough to be applied to processor verification.

A symbolic model checker is one verification tool based on FSM traversal. Given a CTL formula which serves as a specification and an FSM model which represents an implementation, it decides if the CTL formula holds on the FSM. Several designs, such as a pipelined data-path and a DMA controller, have been verified by model checking [Lon93, Mcm93]. However, for processor verification, it is not known how to describe the entire specification of a given processor in CTL. Thus, in this particular target, comparison against a *reference processor*, an unpipelined implementation of a given instruction set, becomes an alternative method. Some research activities [Ael92, Bha94, Bur94] using this approach seem promising.

Even with the latest BDD technologies, it is still difficult to achieve comparison of very large FSMs. Recent advances present two approaches to counter this problem. One is the simplification of the model under verification. This includes abstraction [Lon93], simplification of models under given CTL formulas [Azi94], and uninterpreted evaluation of processor data paths using equation logic [Bur94]. The other approach is to contrive a more efficient verification method dedicated to processors taking certain design information or properties of processors into account. [Bha94] and [Bur94] reduce the costly FSM traversal to several cycles of symbolic simulation using auxiliary information.

In this paper, we propose one method of verifying pipelined processors. The basic verification technology we used is symbolic model checking. The computation cost may be larger than that of [Bha94] and [Bur94] but it requires less auxiliary information. We present a method of achieving comparative verification in the framework of symbolic model checking. Another contribution is a powerful method for simplifying FSMs. We propose a method of achieving uninterpreted evaluation of complex functional units utilizing that only the equivalence of the two processors is of interest.

We experimented with a DLX processor subset and succeeded in exposing several bugs, regarding update conflicts of the program counter, cancellation of prefetched instructions on branch/jump, and pipeline control to avoid data hazard. The verification took from 2 to 15 hours and 40MB of memory on a DECstation 5000/260 with 480MB.

In Section 2, we formulate the verification of processors based on comparison. We present the new FSM abstraction method in Section 3, and show experimental results in Section 4.

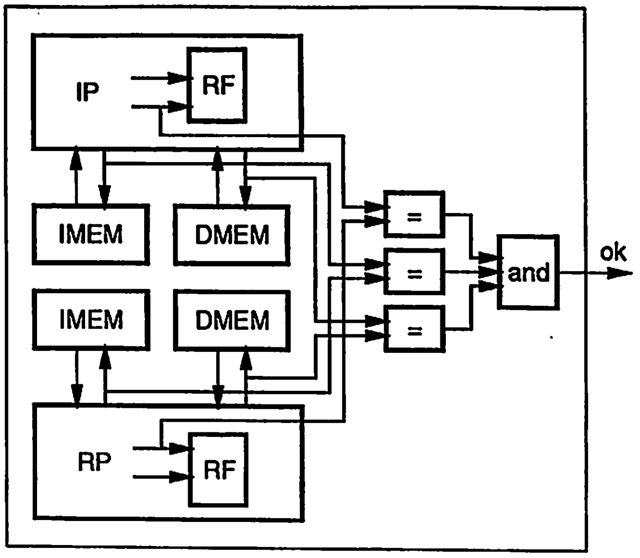


Figure 1: Verification based on comparison.

2 Verification of Processors Based on Comparison

2.1 Reference Processor and Comparison

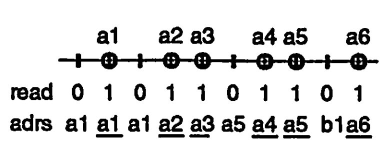
The problem of processor verification is, given a specification and an implementation of a processor and auxiliary information, decide if the implementation meets the specification. There is ambiguity in this definition. We first clarify this point.

Figure 1 shows the rough picture of our verification approach, which is based on comparison. *IP* is an implementation of a processor and *RP* is a reference processor which serves as the specification. The reference processor can be any implementation of the instruction set as long as it satisfies a certain assumption to be given later. The simplest implementation of an instruction set, which executes one instruction per cycle without sophisticated implementation techniques, can serve as the reference processor. Conceptually, we handle a whole processor including an instruction memory IMEM and a data memory DMEM (though the memory will be abstracted later).

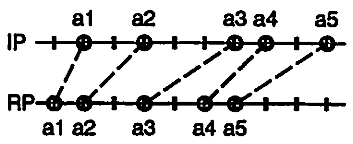
We compare the signal sequences on pairs of certain signal lines for a kind of equivalence. The signal lines to be probed are the read address to the IMEM and the read/write address/data to the DMEM. We could optionally compare the signal sequences on the write address/data to RF (register file) and those on other signal lines, which might expose design errors earlier. The implementation meets the specification if all the pairs of signal sequences are equivalent for arbitrary initial configurations of the IMEMs and the DMEMs. In other words, we investigate if the signal value on line ok in Figure 1 is always 1 for every program and every data.

We will model the whole circuit in Figure 1, including the IP, the RP, the IMEMs, the DMEMs, and the comparator, as a finite state machine (FSM). Then we run a CTL model checker [Lon93, Mcm93] to verify if CTL formula *AG ok* (always globally ok = 1) holds.

There remains two technical difficulties in this verification scheme. One is that equivalence of the signal sequences in the usual sense is not applicable in the above comparison, because



(a) Model of signal sequences.



(b) Equivalence.

Figure 2: Signal sequence.

the IP and the RP can put data on signal lines at different clock cycles. The other difficulty is handling the huge state space of the memories.

We solve the first problem by defining relations between signal sequences that formalize the equivalence in the above sense. We cope with the second problem by replacing each memory by a simpler yet more general model.

2.2 Specification-Implementation Relations of Signal Sequences

2.2.1 Model of Signal Sequences

Figure 2 (a) illustrates an example of signal sequences which a processor outputs to read the IMEM. We are only interested in the values of adrs (address) sent out when the processor is actually in the action of reading the IMEM. Such clock cycles are indicated by the read signal. We will depict these clock cycles by the large dots labeled by the values as in Figure 2. Our goal is to establish that certain relations hold between two signal sequences as shown in Figure 2(b). In order to define the relations formally, we first define some notation for signal sequences.

A *flagged signal sequence* over a domain D is a tuple $s = (a, v)$ where $a \in \mathcal{B}^*$, $v \in D^*$, with $|a| = |v|$. A *filtering function* ϕ is a mapping of flagged signal sequences to D^* defined as

$$\begin{array}{lcll} \phi(\varepsilon, \varepsilon) & = & \varepsilon & \\ \phi(a \cdot b, v \cdot x) & = & \phi(a, v) & \text{if } b = 0 \\ & & \phi(a, v) \cdot x & \text{if } b = 1, \end{array}$$

where ε is the empty sequence, and $\cdot$ denotes concatenation of sequences. Thus ϕ simply executes a masking operation given the flag sequence a.

Assumption: We assume that RP emits the effective signal values in an earlier or the same clock cycle as IP. Let $s_i = (a_i, v_i)$ and $s_r = (a_r, v_r)$ be output sequences of IP and RP, respectively. Let $\sigma[k]$ be the k-th element of a string σ and let $\sigma[k : l] = \sigma[k] \cdot \sigma[k+1] \cdots \sigma[l]$ (ε if $k > l$). Let $\#_b(\sigma)$ be the number of b's in σ. Then this assumption is formally written as

$$\forall k : \#_1(a_i[1 : k]) \leq \#_1(a_r[1 : k]).$$

The formulation above is similar to that of [Ael92], but here the sequence a of $s = (a, v)$ is not derived from the design but originally emitted by the processor. This will cause a difference in how prefetched instructions are cancelled.

2.2.2 Specification-Implementation Relations

We define two relations, called *specification-implementation (SI) relations of type-ii and type-ir*, which are designed for verification of pipelined processors.

Definition 1 Let s_r and s_i be flagged sequences. Then, the SI relation of signal sequences of type-ii, denoted $SI_{ii}(s_r, s_i)$, is defined as:

$$SI_{ii}(s_r, s_i) \Leftrightarrow \phi(s_r) = \phi(s_i).$$

$\square$

Figure 3 (a) illustrates the SI relation of type-ii. The relation holds if the same effective data are sent out in the same order, even if they do not agree in the clock cycles.

The other relation is related to the cancellation of prefetched instructions in pipelined processors. If the pipelined processor is implemented based on the "predict-not-taken" scheme [HP90], the processor may fetch more instructions than it actually executes. As long as we observe `adrs` and `read` lines, the output sequence of the pipelined processor may contain more effective read addresses than for RP. The *specification-implementation relation of type-ir* is defined so that the output sequences will be judged to be equivalent in the presence of irrelevant instructions.

Definition 2 Let s_r and s_i be flagged signal sequences, and $P_k \subset \mathcal{B}^*$ be the set of all Boolean strings that contain no more than k consecutive 0s. Then, the SI relation of signal sequences of type-ir and degree k, denoted $SI_{ir(k)}(s_r, s_i)$, is defined as:

$$SI_{ir(k)}(s_r, s_i) \Leftrightarrow \exists d \in P_k : \phi(s_r) = \phi(d, \phi(s_i)).$$

$\square$

Figure 3 (b) shows an instance of this type of SI relation, where `b1` is the irrelevant data and the two sequences satisfies the SI relation by ignoring `b1`.

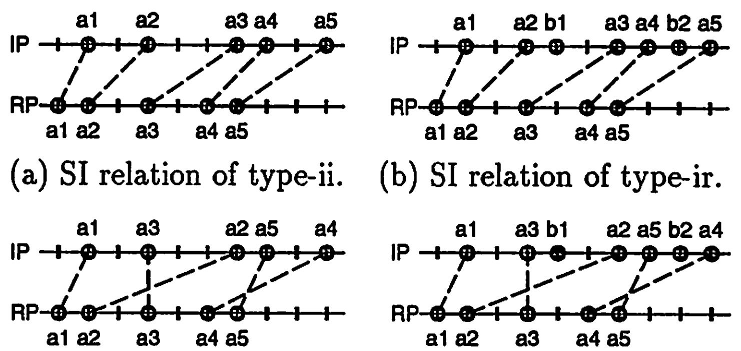
The two SI relations deal with the case where IP generates the effective values in the same order as RP (in-order). We could define similar SI relations for the case where the order is not the same (out-of-order) as illustrated in Figure 3 (c) and (d), though we do not handle these in this paper.

2.2.3 Equivalence Checking

As shown in Figure 1, the SI relation of each pair of signal sequences is checked by a module which outputs 1 if and only if the sequences given so far satisfies the SI relation. We show in this section that the modules that check the SI relation of type-ii and type-ir can be implemented as FSMs.

Basically, the type-ii relation is checked by using a queue; every time the faster processor (the RP) outputs an effective data, it is enqueued, and every time the slower processor (the IP) produces an effective data, it is compared with the dequeued data.

However, this simple scheme fails due to possible explosion of the queue size; the queue length is not necessarily bounded. Suppose RP gives a result every clock cycle while IP takes



(a) SI relation of type-ii. (b) SI relation of type-ir.

(c) SI relation of type-oi. (d) SI relation of type-or.

Figure 3: Specification-Implementation (SI) relations.

two clock cycles to output each result due to data hazards. Then the length of the queue increases every other clock cycle and infinite capacity is required.

This problem can be solved by *freezing* RP when necessary. We use a checking module with a queue of a finite size. When the queue becomes full, the checking module emits a `freeze` signal to the RP. The `freeze` signal stops the operation of RP temporarily by disabling the clock to RP. Since IP keeps running, a dequeue operation eventually takes place; then the `freeze` signal is turned off. Thus checking can be achieved with finite state.

Another problem is that we may overlook that IP stuck in a idle loop. Since the checker only examines prefixes of the two sequences, it yields 1 even if the IP does not produce any effective data. This is also easily solved by providing a counter. Each time IP emits an ineffective data, the counter is incremented. If the value of the counter exceeds a predetermined limit, then "ok" is set to 0.

Figure 4 shows the algorithm to check the SI relation of type-ii. Initially the queue `Q` is empty and the counter `idle` is 0. At every clock cycle, the output value of `ok` and `freeze` is computed. If the value of a_r at the clock cycle (expressed as `a_r`) is 1, the corresponding value (`v_r`) is enqueued. In case the value of a_i at the clock cycle (`a_i`) is also 1, the value of v_i (`v_i`) is compared with the head element of the queue to evaluate `ok`. If `a_i` is 0, the counter is incremented and, if it exeeds a limit (`idle_LIMIT`), `ok` is set to 0. The freeze signal (`freeze`) is set to 0 if the queue is full.

The type-ir checker is obtained by simply revising line 12 of the algorithm of type-ii. Disagreement of the effective values is allowed for at most k times in succession. We only have to provide a counter to count the disagreement. The algorithm is shown in Figure 5.

This algorithm for type-ir is not complete because the algorithm investigates only one possible match when there are multiple choices. For example, suppose the following data is given for $k = 2$.

$$\begin{array}{lcl} a_r &=& 1\ 1 \\ v_r &=& 3\ 4 \end{array} \qquad \begin{array}{lcl} a_i &=& 1\ 1\ 1\ 1\ 1 \\ v_i &=& 3\ 3\ 7\ 8\ 4 \end{array}$$

There are two candidates $v_i[1] = 3$ and $v_i[2] = 3$ that match with the first signal $v_r[1] = 3$. If we choose $v_i[1]$ to match with $v_r[1]$, then it is impossible to match $v_r[2] = 4$ with $v_i[5] = 4$

```
initial {
 Q = new queue;
 idle = 0;
}

always clock {
  if (a_r) Q.enq(v_r)
  if (a_i) {
    idle = 0;
    if (!Q.empty()) {
      v = Q.deq();
      if (v_i==v) {ok = 1}
      else        {ok = 0}
    }
    else {ok = 0}
  }
  else {ok = (++idle<=idle_LIMIT)}
  freeze = Q.full();
}
```

Figure 4: Algorithm of the checking module of type-ii.

```
initial {
 Q = new queue;
 idle = 0;
 disagree = 0;
}

always clock {
  if (a_r) Q.enq(v_r)
  if (a_i) {
    if (!Q.empty()) {
      idle = 0;
      v = Q.deq();
      if (v_i==v) {ok = 1; disagree = 0}
      else        {ok = (++disagree<=k)}
    }
    else {ok = 0;}
  }
  else {ok = (++idle<=idle_LIMIT)}
  freeze = Q.full();
}
```

Figure 5: Algorithm of the checking module of type-ir.

because there are more than $k = 2$ irrelevant data between $v_i[1]$ and $v_i[5]$. This would be the result of the algorithm in Figure 5, the checking module would conclude erroneously that the two sequences don't satisfy the SI relation, even though the matching of ($v_r[1] = 3$, $v_i[2] = 3$) and ($v_r[2] = 4$, $v_i[5] = 4$) satisfies the relation.

A sufficient condition for the algorithm to work correctly is as follows.

Proposition 1 The algorithm of Figure 5 judges the SI relation of type-ir and degree k correctly if there is no repeated data value in any substring of length $k+1$ in $\phi(s_i)$.
[Proof] If the assumption holds, there is only one candidate element in s_i that matchs with each effective datum in s_r even if the consecutive effective data in s_r have the same values. □

Later, in Section 4.4, we show by example how this assumption can be forced to hold.

3 Reduction of Model Size Based on Uninterpreted Evaluation

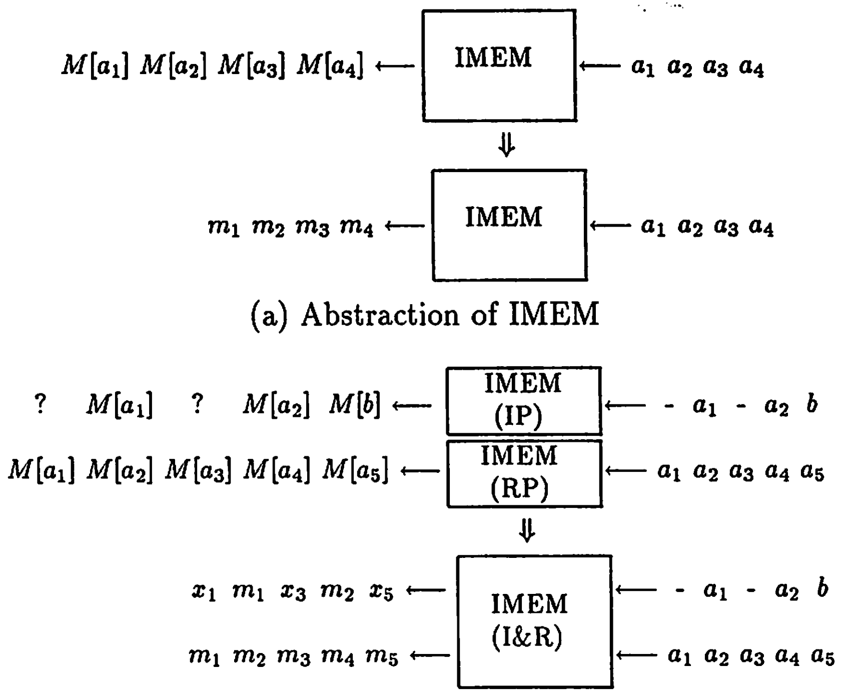
Since IP, RP, the IMEMs, the DMEMs and the checking modules are all modeled as finite state machines, it is theoretically possible to decide the correctness of the implementation by CTL model checking. However, the model is too big for state traversal. If the address is 32 bits wide, for example, even a single word has 2^{32} states. Register files can contain 256 words, and processors can contain multipliers and floating-point units.

To deal with larger and more complex FSM models, several methods of simplifying the FSMs are proposed. One is to reduce the size of the FSMs, taking the given temporal formula into account, so that the result of the verification will be exactly the same [Chi92, Azi94]. Another approach is to simplify the FSMs (and also the CTL formula) allowing *false negatives*, where the implementation is correct but the verification reports it is not. There are variety of simplification methods of this type [Lon93]. They are referred to as conservative *abstractions*. We propose a new conservative simplification method which uses the fact that our two processors have similar behavior. It uses *uninterpreted evaluation*; namely, it attempts to prove the correctness of the implementation without evaluating the details of memory modules or functional units. It consists of a series of two simplification steps. First we replace the output sequence of each module targeted for simplification by a nondeterministic sequence. Then we reduce the number of bits to represent the data. We refer to the first step as an *N1 abstraction* and the second as an *N2 abstraction.*

3.1 Abstraction of Memories

3.1.1 Basic Concept

Figure 6 (a) shows the fundamental idea of the N1 abstraction. It is based on replacement of the output sequence of a module by a nondeterministic sequence. IMEM outputs $M[a_1]M[a_2]M[a_3]M[a_4]$ in response to an input sequence $a_1a_2a_3a_4$, where $M[a]$ represents the contents of IMEM at location a. For the abstraction, we replace the output sequence by a nondeterministic sequence $m_1m_2m_3m_4$, where each m_i can take any value from its domain. This replacement drastically reduces the number of the state variables of IMEM since there is no need to maintain the values for the address space.



$M[a_1]\ M[a_2]\ M[a_3]\ M[a_4]$
IMEM
$a_1\ a_2\ a_3\ a_4$
$m_1\ m_2\ m_3\ m_4$
IMEM
$a_1\ a_2\ a_3\ a_4$
(a) Abstraction of IMEM
? $M[a_1]$? $M[a_2]$ $M[b]$
IMEM (IP)
- a_1 - a_2 b
$M[a_1]\ M[a_2]\ M[a_3]\ M[a_4]\ M[a_5]$
IMEM (RP)
$a_1\ a_2\ a_3\ a_4\ a_5$
$x_1\ m_1\ x_3\ m_2\ x_5$
$m_1\ m_2\ m_3\ m_4\ m_5$
IMEM (I&R)
- a_1 - a_2 b
$a_1\ a_2\ a_3\ a_4\ a_5$
(b) Abstraction of pairs of IMEM

Figure 6: Replacement of a memory module by a nondeterministic sequence.

The drawback of this simplification is conservative; the verifier may report a false negative but never report a false positive. This is because we investigate superfluous cases in addition to the original ones. Since the given specification is *AG ok*, the addition of superfluous cases disallows false positives.

Suppose an input sequence to IMEM consists of addresses that are different from each other, $a_1a_2a_3a_4$ = 32 36 40 44, for example. Since we must achieve verification for every possible initial configuration of IMEM, the output sequence corresponding to the input sequence can be any sequence, $M[a_1]M[a_2]M[a_3]M[a_4]$ = 100 176 287 153 or 273 273 222 103 etc. The nondeterministic sequence $m_1m_2m_3m_4$ can be also any sequence. Therefore, in this case of unique addresses, there exists a one to one correspondence between the original output sequence and the nondeterministic sequence. Hence there is no loss of accuracy (no false negatives) in the verification results.

The penalty of abandoning the memory cells occurs when there are repeated addresses in the input sequences. Suppose $a_1a_2a_3a_4$ = 32 32 34 38. The first and the second values in $M[a_1]M[a_2]M[a_3]M[a_4]$ are then the same. This does not hold in the nondeterministic sequence, and hence this replacement is not exactly accurate. However, note that it does not lead to the incompleteness of verification, because the first and the second values in the nondeterministic output sequence can be the same and all the possible original output sequences are completely covered.

In our verification application, we deal with two processors which access two IMEMs of identical contents. Figure 6 (b) illustrates the simplification for this situation. The two processors are supposed to access the IMEMs by the same address sequences in the sense of type-ii

```
 1: initial {
 2:  Q = new queue;
 3:  idle = 0;
 4: }

 5: always clock {
 6:   if (a_r) {y_r = new; Q.enq((v_r,new))}
 7:   if (a_i) {
 8:     idle = 0;
 9:     if (!Q.empty()) {
10:       (v,y) = Q.deq();
11:       if (v_i==v){ok = 1; y_i = y}
12:       else       {ok = 0}
13:     }
14:     else {ok = 0}
15:   }
16:   else {ok = (++idle<=idle_LIMIT)}
17:   freeze = Q.full();
18: }
```

Figure 7: Algorithm of the matching module of type-ii.

or type-ir relations. If the two addresses matches, then the same nondeterministic value is assigned to both of the corresponding data. Otherwise, nondeterministic values are generated for each separately.

The same discussion concerning conservativeness as in the single IMEM case holds also in this case. This simplification does not reflect the fact that the same addresses may be given to the IMEMs more than once but it is still conservative since the set of the behaviors of the simplified model includes those of the original.

3.1.2 Matching Module

The module that replaces the two IMEMs in Figure 6 (b) is also implemented as an FSM. Actually it is an extension of the checking module.

Figure 7 shows an algorithm for the matching module of type-ii. The only differences from the simple checking module are in lines 6, 10, 11, and 12. When `a_r = 1` (line 6), a nondeterministic value is assigned to `y_r` and inserted into the queue along with the input data `v_r`. When `a_i = 1` and the queue is not empty (lines 10, 11, and 12), the output value stored in the queue is emitted to `y_i`.

Similarly, the algorithm of the matching module of type-ir is derived from that of the checking module, as shown in Figure 8 .

3.1.3 Handling of Data Memory

We have discussed IMEM which is read only, while both read and write operations are provided with the DMEM. Figure 9 illustrates a way of handling a read/write memory. It is split into the read part and the write part. The read part is the same as the IMEM. The write part is

```
initial {
 Q = empty_queue;
 idle = 0;
 disagree = 0;
}

always clock {
  if (a_r) {y_r = new; Q.enq((v_r,new))
  if (a_i) {
    if (!Q.empty()) {
      idle = 0;
      (v,y) = Q.deq();
      if (v_i==v) {ok = 1; disagree = 0; y_i = y   }
      else        {ok = (++disagree<=k); y_i = new2}
    }
   else {ok = 0}
  }
  else {ok = (++idle_i<=idle_LIMIT)}
  freeze = Q.full();
}
```

Figure 8: Algorithm of the matching module of type-ir.

only a checking module that checks the SI relation between the sequences of write addresses and write data.

This separated model introduces a conservative inaccuracy where the effects of write operations are not reflected in the read operations. However, the verification results are accurate as long as IP and RP access DMEM in the same order. As for conservativeness, the same discussion as in the IMEM case holds also in this case.

3.2 Abstraction of Data Path

We can also apply the N1 abstraction to data paths. Figure 10 illustrates an example of the abstraction on a multiplier where y_i and z_i are nondeterministic sequences. The same

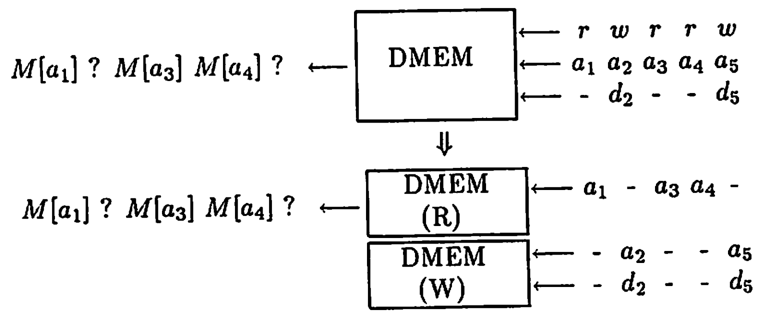


Figure 9: Handling of an R/W memory.

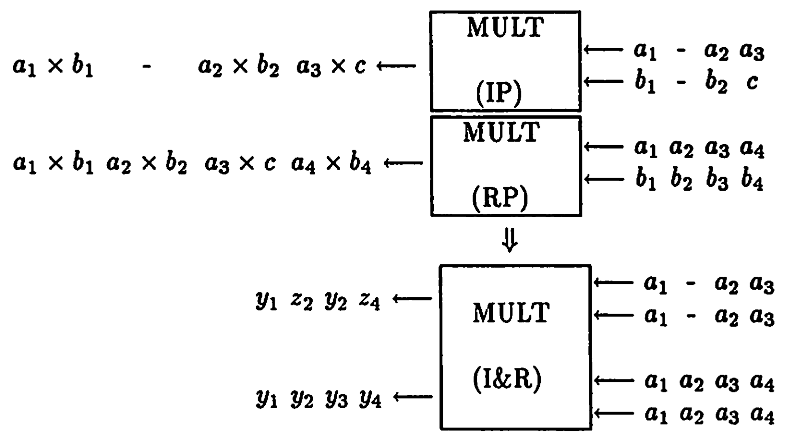


Figure 10: Abstraction of multipliers.

nondeterministic output values are generated, if and only if the operands are the same. The output sequence of the multiplier of RP is replaced by a nonderterministic sequence y_1, y_2, y_3, y_4, $\cdots$. The first output value of the IP multiplier ($a_1 \times b_1$) is equal to the first output value of the RP multiplier, so it is replaced by the same nondeterministic value y_1. Similarly the third output of the IP multiplier is replaced by y_2. On the other hand, the second and the fourth outputs of the IP multiplier do not have corresponding values in the output sequence of the RP multiplier, so they are replaced by other nondeterministic values z_2 and z_4.

By the same discussion as for IMEM, the application of N1 abstraction to the functional units without memory elements is conservative.

Although the information of the input values themselves is totally destroyed, there is equivalence between the values in the output sequences of RP and IP. Thus there will be no false negatives as long as the SI relation between the two processors does not depend on a property of the operation achieved by the unit. For example, suppose RP and IP both compute $a \times b$. We can judge if both processors compute the equivalent result without evaluating the multiplier. On the other hand, $2 \times x$ is the same as $x + x$ or a shift, but these data will be judged different, possibly causing a false negative depending on the property being checked. Another example possibly causing a false negative is when RP computes $a \times b \times c$ by computing $a \times b$ first and then multiplying by c while IP computes $b \times c$ first and then multiplies by a.

In general, as long as both processors produce the new data in exactly the same way, there are no false negatives. All this is equivalent to the assumption that the functional units are correctly implemented, something that can be verified separately.

3.3 Reduction of the Number of Bits

Once functional units are simplified by the N1 abstraction, we can reduce the number of bits (N2 abstraction) to represent the data *without further losing verification accuracy.* We first present the theorems and then prove them in the rest of this section.

Theorem 2 If all the functional units associated with a data type D are replaced by their N1 abstractions and all the matching and checking modules are of type-ii, then exactly the same

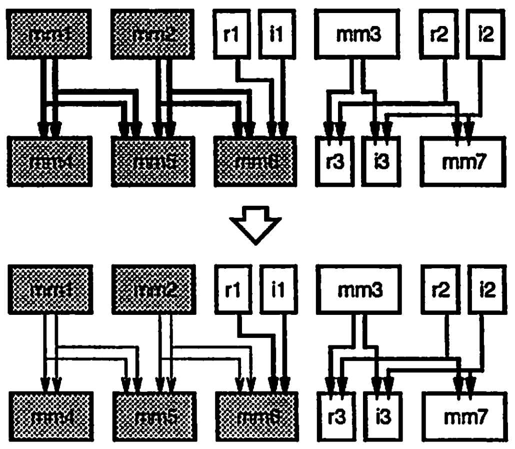


Figure 11: N2 abstraction.

verification result is obtained even if the number of bits to represent D is reduced into one bit.
$\square$

Theorem 3 If all the functional units associated with a data type D are replaced by their N1 abstractions and the whole model contains matching or checking modules of type-ir of at most k and the assumption of proposition 1 holds, then exactly the same verification result is obtained even if the number of bits to represent D is reduced to $\lceil \log(k+2) \rceil$ bits. $\square$

[Proof of Theorem 2] Figure 11 shows the block diagram of the FSM on which the N2 abstraction is to be applied. The signal lines of data type D are depicted by bold lines, each of which is implemented as d bit binary signal lines. The condition where we can apply the N2 abstraction is that all the modules connected to the signal lines of data type D should be checking/matching modules of type-ii. Then the claim is that the verification result is the same even if we replace all the bold lines of d bits by signal lines of 1 bit.

Denote the original model and the simplified model as M and M', respectively. $N \models \eta$ expresses that a CTL formula η is true for a model N. Let ok_m be the ok signal of the m-th checking/matching module associated with data type D. The "verification result is the same" is stated as, for all m,

$$M \models AG\ ok_m \text{ iff } M' \models AG\ ok_m. \tag{1}$$

We investigate how $AG\ ok_m$ is represented. For simplicity, consider a matching module which has only one pair of inputs (from RP and IP) of type D. The same discussion holds for the general case. Let s_r and s_i be the signal sequences over D fed into the m-th checking/matching module. Then $AG\ ok_m$ is true iff the SI relation holds between s_i and s_r of arbitrary length for any assignment determined by the preceding matching module. Let $|s_i| = n$ and $|\phi(s_i)| = l_{s_i}(n)$. Then,

$$M \models AG\ ok_m \quad \text{iff} \quad \forall n : \bigwedge_{k=1}^{l_{s_i}(n)} \phi(s_r)[k] = \phi(s_i)[k] \tag{2}$$
for any assignment by the preceding matching module.

The preceding matching module assigns all possible values in D to $\phi(s_r)[k]$ and $\phi(s_i)[k]$ under the constraint that the values are the same if the corresponding input matches and the value should be the same. We can express this constraint using variables $y_1, \cdots, y_n$ and $z_1, \cdots, z_n$ as was shown in Figure 10. For notational convenience, let us use $y_{n+1}, \cdots, y_{2n}$ in the place of $z_1, \cdots, z_n$. The constraint is expressed as

$$\phi(s_r)[k] = y_{r_k} \text{ and } \phi(s_i)[k] = y_{i_k}. \tag{3}$$

where, $1 \leq r_k \leq 2n$, $1 \leq i_k \leq 2n$. Then (2) is written as

$$\begin{aligned} M \models AG\ ok_m \quad &\text{iff} \quad \forall n, \forall(y_1, \cdots, y_{2n}) \in D^{2n} : \bigwedge_{k=1}^{l_{s_i}(n)} \phi(s_r)[k] = \phi(s_i)[k] \\ &\text{iff} \quad \forall n, \forall(y_1, \cdots, y_{2n}) \in D^{2n} : \bigwedge_{k=1}^{l_{s_i}(n)} y_{r_k} = y_{i_k}. \end{aligned} \tag{4}$$

A similar transformation applies for the $M' \models AG\ ok_m$, except that M' handles domain $\mathcal{B} = \{0, 1\}$ instead of D. Then we get the following result.

$$M' \models AG\ ok_m \quad \text{iff} \quad \forall n, \forall(y_1, \cdots, y_{2n}) \in \mathcal{B}^{2n} : \bigwedge_{k=1}^{l_{s_i}(n)} y_{r_k} = y_{i_k}. \tag{5}$$

Now, we only have to show the right hand sides of (4) and (5) are identical in order to prove $M \models AG\ ok_m$ iff $M' \models AG\ ok_m$. This is easily proved by the following lemma.

Lemma 4 For any two domains D_1 and D_2 and for any given series $a_1, a_2, \cdots, a_K$ and $b_1, b_2, \cdots, b_K$ where $1 \leq a_k \leq N$ and $1 \leq b_k \leq N$ (for $k = 1, 2, \cdots, N$),

$$\forall(x_1, \cdots, x_N) \in {D_1}^N : \bigwedge_{k=1}^{K} x_{a_k} = x_{b_k} \text{ iff } \forall(x_1, \cdots, x_N) \in {D_2}^N : \bigwedge_{k=1}^{K} x_{a_k} = x_{b_k}. \tag{6}$$

[Proof] Both are true iff $a_1 = b_1, a_2 = b_2, \cdots, a_K = b_K$. □

[Proof of Theorem 3] Since the assumption of Proposition 1 holds (no repeated data values), then $M \models AG\ ok_m$ and $M' \models AG\ ok_m$ are also reduced to (4) and (5), respectively, because only one matching between s_r and s_i is possible.

In order to find a match for an element in s_r, we must test at most $k+1$ elements in s_i. When a match to this element in s_r is not found in these elements, the values of all of those $k+2$ elements are different. Thus we must be able to distinguish $k+2$ elements to correctly decide if a match is possible. $\lceil \log(k+2) \rceil$ bits satisfy this requirement. □

Note that $\lceil \log(k+2) \rceil$ bits only guarantee a sufficient amount of information to achieve exact verification under this assumption of proposition 1. In order to make the assumption hold, we need to contrive some means for this. We will show how to do this by an example in Section 4.4.

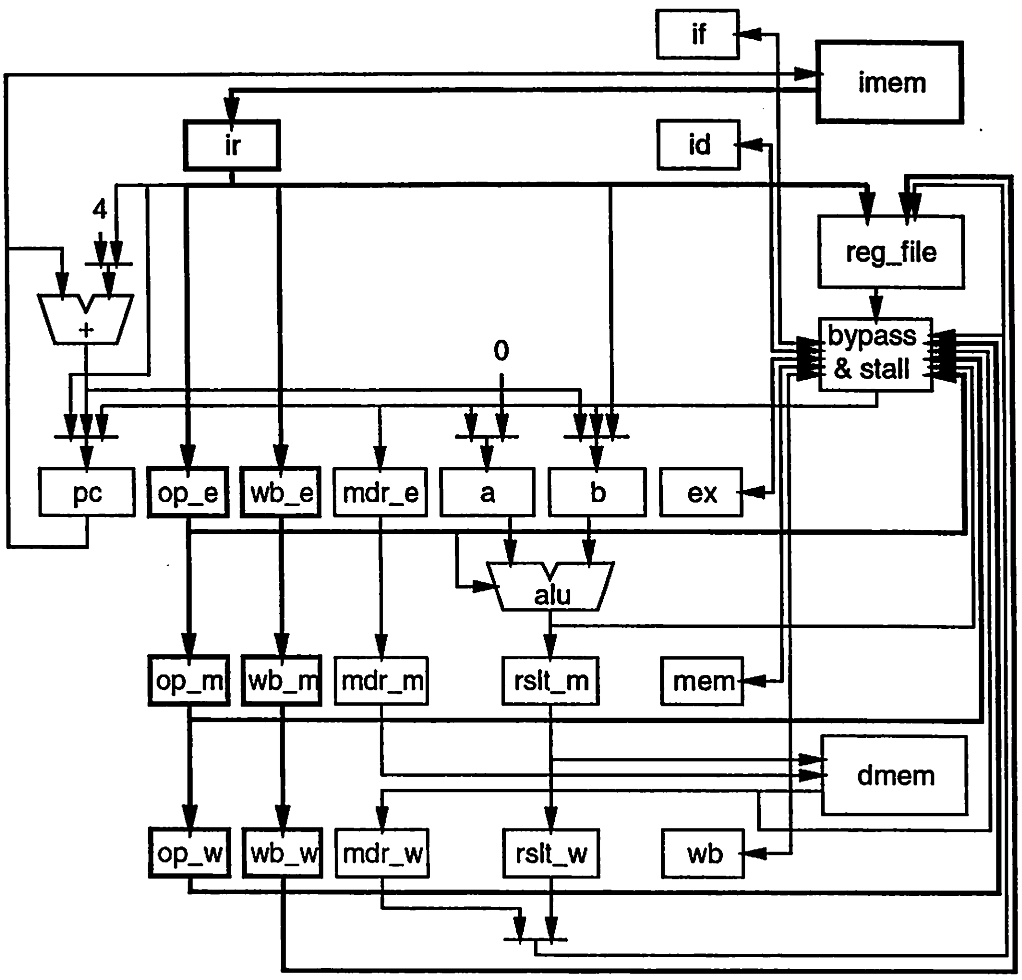


Figure 12: Block diagram of the IP.

4 Experimental Results

We applied the verification method to an implementation of a subset of the DLX processor [HP90]. The processor is named the FDDP and the design was written in a hardware design language SFL [Nak91].

The implementation is based on a standard 5-stage pipeline (IF: instruction fetch, ID: instruction decode, EX: ALU execution, MEM: memory access, and WB: register write back). It does not provide floating-point operations. Neither exception nor interrupt is implemented, but a bypass circuit to avoid pipeline stall is provided.

4.1 Simplification of the Model

Figure 12 is the block diagram of the IP. We applied the N1 and N2 abstractions to the IMEM, the DMEM, the ALU and the adder of the program counter. As a result, the bit-width of the signal lines are reduced to 1 bit except for the ones depicted by bold lines in the figure.

The abstraction also reduces the size of the instruction set, because the instructions which

Table 1: Reduced instruction set of the processor.

op code		function
ALU	op, r_1, r_2, r_w	`pc<-pc+4, R[`r_w`]<-R[`r_1`]`op`[`r_2`],`
ALUI	op, r, imm, r_w	`pc<-pc+4, R[`r_w`]<-R[`r_1`]`$op\ imm$`,`
LOAD	r, adr	`pc<-pc+4, R[`r`]<-M[`adr`],`
STORE	r, adr	`pc<-pc+4, M[`adr`]<-R[`r`],`
BEQZ	$r, dspl$	`if (R[`r`]==0) pc<-pc+4+`$dspl$ `else pc<-pc+4,`
BNEZ	$r, dspl$	`if (R[`r`]!=0) pc<-pc+4+`$dspl$ `else pc<-pc+4,`
J	$adrs$	`pc<-`$adrs$`,`
JAL	$adrs$	`pc<-`$adrs$`, R[1]<-pc+4,`
JR	$adrs$	`pc<-R[`r`],`
JALR	$adrs$	`pc<-R[`r`], R[1]<-pc+4,`

Table 2: Size of the model.

	N1 (Memory)	N1(OU) & N2	Aprx-1	Aprx-2
IP	1317	85	39	39
RP	1220	32	4	4
MM	59	76	50	13
Total	2572	193	93	56

gives the same result under the abstraction are grouped into one. For example, all the ALU instructions with register operands become a single ALU instruction. Table 1 shows the reduced instruction set of the processor.

Table 2 shows the size of the whole FSM model for verification in terms of the number of state variables (bits). The column "N1 (Memory)" is the model after applying the N1 abstraction to the IMEMs and the DMEMs and the column "N1(OU) & N2" is the result of additionally applying the N1 abstraction to the functional units and then the N2 abstraction to all appropriate signal lines.

The size 193 of the model, even after all the abstraction, is still too large for model checking. So we applied other simplifications. The first is the reduction of the register file size. The original design, with 32 registers, was reduced to 4 registers. The second is the simplification of the functional units. Since even the matching modules require large computation cost, some functional units are replaced by simpler models instead of the matching modules as shown in Figure 13. Here the circled plus signs stand for exclusive-or. Note that these simplifications are not guaranteed to be conservative anymore and may cause false positives.

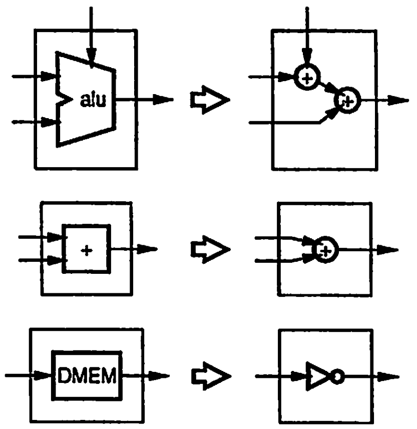


Figure 13: Simplification of operational units.

4.2 Verification Tool

As for the verification tool, we used *SMV* developed at CMU [Mcm93, Lon93]. It is a CTL model checker based on the implicit state traversal method using BDDs. We used the version with dynamic variable ordering of the BDDs [Che94]. We also used the partitioned image computation (-cp option) and incremental image computation (-inc). Without these options, the verification could not finish. All experiments are conducted on DECstation 5000/260 with 480MB of memory.

4.3 Bugs Found by Verification

Several bugs in the implemented processor were discovered during the experiments. The following is a summary of the experiments.

1. Mutual exclusion of PC update

   A bug was found in the first version of the implementation related to the mutual exclusion of PC update. PC is updated in the IF stage on ALU and load/store instructions and in the ID stage on branch/jump instructions. If an operation in the second category is followed by one in the first category (BEQZ followed by ALU, for example), the PC update of the branch/jump operation should have the higher priority. However in the first version, the priority was reversed. This was a trivial bug caused during the translation of the original description into the FSM model of SMV.

   Only type-ii matching modules are required for this verification. It took about 2 hours using 12MB of memory.

   We fixed the bug and continued the verification.

2. Disagreement of specification of branch

   It was found that cancellation of prefetched instruction on a jump/branch instruction was not implemented in the IP. It took about 4 hours and 12MB of memory using type-ii checking/matching modules.

We continued with two streams of experiments. One was to correct the specification (the RP) to realize delayed branches. The other was to redesign the IP so that it cancels prefetched instructions on branch and jump instructions.

3. Data hazard

   On the delayed branch version, another disagreement of the two behaviors was detected. When a LOAD instruction loads data to a register and the next ALU instruction refers to the same register, the second instruction fetches invalid data. Even though the bypass logic is correct and the data hazard between two consecutive ALU instructions is detected, the ALU instruction right after the LOAD instruction should be stalled at least one clock cycle. It took about 4 hours and 12MB of memory using type-ii checking/matching modules to find such an error trace.

   This can be a correct implementation, if it is assumed that the compiler always inserts an NOP right after LOAD instructions.

   Nevertheless, we redesigned the IP to stall for one clock cycle if this situation is detected and ran verification again. This time, it passed verification. It took about 2.5 hours and 18MB of memory.

4. Bugs in pipeline control logic

   On the prefetch-cancel version, another bug was found. There was an error in a stage control logic where the IP stalls forever on LOAD followed by BEQZ. In order to handle the prefetch-cancel implementation, a matching module of type-ir is required. It took about 15.5 hours and 40MB of memory for verification to find this bug.

   After this bug was fixed the implementation passed the verification, taking about 4.6 hours and 27MB of memory. It seems that the previous verification consumes much more computation resource because an error trace (a sequence of states that leads to the violation of the given CTL formula) had to be computed after deciding NG (no good).

4.4 Problem of False Matching

In the experiments on the prefetch-cancel version, we had to add auxiliary logic to avoid false matching of the type-ir matching module. (See proposition 1).

Figure 4.4 (a) shows an example of the situation. The branch/jump destination happened to be the same as when branch/jump was not taken. In this case, the address 32 is once fetched and cancelled in the IP. Thus the address 32 of the RP should be matched with the next 32. However, in the algorithm described in Section 2, the match is established between the first elements available. Then, the data m_3 for address 32 is thrown away in the IP and the false data x_4 is fed into the IP to cause a disagreement in the behaviors.

This problem is solved by putting a *tag* on the address data as shown in Figure 4.4 (b). The tag $_B$ indicates that the address is generated by the branch/jump instructions. In this case, the correct matching takes place, since the prefetched data does not have the tag. Instead, two bits (one for the address data and one for the tag) are required.

```
24  ALUI
28  J      32
32  ALU
    ...
```

	m_1	m_2	~~m_3~~	x_4	m_4	
IP:	24	28	~~32~~	32	36	$\cdots$
RP:	24	28	32	36	40	$\cdots$
	m_1	m_2	m_3	m_4	m_5	

(a) False matching.

	m_1	m_2	~~x_3~~	m_3	m_4	
IP:	24	28	~~32~~	$_B$32	36	$\cdots$
RP:	24	28	$_B$32	36	40	$\cdots$
	m_1	m_2	m_3	m_4	m_5	

(b) Correct matching using tags.

Figure 14: Problem of false matching.

5 Conclusion

We presented a method of processor verification based on comparison of an implemented processor with a reference processor. We succeeded in reducing the computation cost by using the strong abstraction method based on uninterpreted evaluation of memories and complex modules. Experimental results on a subset of the DLX processor were given.

In order to deal with practical processor designs, there are still many difficulties to be resolved. We must be able to handle more sophisticated architectures that allow multiple instruction issues and out of order execution. We must also deal with exceptions and interrupts.

The huge computation cost is the biggest challenge. It seems to be difficult with FSM based approaches to handle larger models than a subset of the DLX. One possibility is to combine uninterpreted evaluation using BDDs with symbolic simulation [Bur94, Bha94].

Acknowledgement

The authors would like to express sincere appreciation to Prof. I. Shirakawa of Osaka University who offered the chance to pursue this new research topic. We would also like to thank to R. Hojati, Y. Kukimoto, T. Nakata, A. Takahara, Y. Watanabe, and all the members of the CAD group of UC Berkeley.

References

[Azi94] A. Aziz, T. R. Shiple, V. Singhal, A. L. Sangiovanni-Vincentelli: "Formula-dependent equivalence for compositional CTL model checking," in *Proc. International Conference on*

Computer-Aided Verification, LNCS 818, pp. 324–337 (June 1994).

[Ael92] F. V. Aelten: "Automatic procedures for the behavioral verification of digital designs," *Ph. D. dissertation*, Dept. Electrical Engineering and Computer Science, MIT (May 1992).

[Bea94] D. L. Beatty and R. E. Bryant: "Formally verifying a microprocessor using a simulation methodology," in *Proc. ACM/IEEE 31st Design Automation Conf.*, pp. 596–602 (June 1994).

[Bha94] V. Bhagwati and S. Devadas: "Automatic verification of pipelined processors," in *Proc. ACM/IEEE 31st Design Automation Conf.*, pp. 603–608 (June 1994).

[Bry86] R. Bryant: "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 677–691 (Aug. 1986).

[Bur94] J. Burch and D. Dill: "Automatic verification of pipelined microprocessor control," in *Proc. International Conference on Computer-Aided Verification* (June 1994).

[Che94] K. C. Chen and C. Coelho: "A system for the Formal Verification of Verilog Descriptions ," submitted to Euro-DAC (1994).

[Chi92] M. Chiodeo, T. R. Shiple, A. L. Sangiovanni-Vincentelli, and R. K. Brayton: "Automatic compositional minimization in CTL model checking", in *Proc. Int'l Conf. on Computer-Aided Design*, pp. 172-178 (Nov. 1992).

[Cyr93] D. Cyruluk: "Microprocessor verification in PVS: A methodology and simple example," Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory (Dec. 1993).

[HP90] J. L. Hennessy and D. A. Patterson: *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, CA, USA (1990).

[Lon93] D. E. Long: "Model checking, abstraction and compositional verification," CMU-CS-93-178, School of Computer Science, Carnegie Mellon University (Ph. D. Dissertation), (June 1993).

[Mcm93] K. McMillan: *Symbolic Model Checking*, Kluwer Academic Publishers (1993).

[Nak91] R. Camposano and W. Wolf (Ed.): *High-Level VLSI Synthesis*, Chapter 9, pp. 205–229, Kluwer Academic Publishers, Boston (1991).

[Rud93] R. Rudell: "Dynamic variable ordering for ordered binary decision diagrams," in *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*, pp. 42–47 (Nov. 1993).

[Tou90] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli: "Implicit state enumeration of finite state machines using BDD's," in *Proc. IEEE Int'l Conf. on Computer-Aided Design*, pp. 130–133 (Nov. 1990).