Formal Methods in System Design, 20, 159–186, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

# Verification of Out-Of-Order Processor Designs Using Model Checking and a Light-Weight Completion Function

SERGEY BEREZIN AND EDMUND CLARKE
*Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA*

ARMIN BIERE
*Dept. of Computer Science, Institute of Computer Systems, ETH, Zürich*

YUNSHAN ZHU
*Advanced Technology Group, Synopsys, Inc., Mountain View, CA, USA*

**Abstract.** We present a new technique for verification of complex hardware devices that allows both generality and a high degree of automation. The technique is based on our new way of constructing a "light-weight" completion function together with new encoding of uninterpreted functions called *reference file representation*.

Our technique combines our completion function method and reference file representation with compositional model checking and theorem proving. This extends the state of the art in two directions. First, we obtain a more general verification methodology. Second, it is easier to use, since it has a higher degree of automation.

As a benchmark, we take Tomasulo's algorithm for scheduling out-of-order instruction execution used in many modern superscalar processors like the Pentium-II and the PowerPC 604. The algorithm is parameterized by the processor configuration, and our approach allows us to prove its correctness in general, independent of any actual design.

**Keywords:** Tomasulo's algorithm, formal verification, model checking, theorem proving, reference file, completion function

## 1. Introduction

Modern microprocessors have become extremely complicated, involving superscalar pipelines and *out-of-order* (OOO) instruction execution [12], and a new generation of VLIW and microcode-based designs. This complexity increases the demand for fast but reliable validation methods to ensure timely delivery of new chips to the market with as few errors as possible.

Formal verification is one of the most precise techniques that can guarantee the correctness of the design. However, the growing complexity of microprocessors makes formal verification increasingly difficult because data and control flow are tightly coupled. Therefore, a formal model cannot easily separate the data and focus only on control flow, or vice versa. Instead, it has to capture all the relevant data dependencies in each control state. As a result, the state space may become enormous.

In this paper, we consider verification techniques for a particular type of OOO processors which employ Tomasulo's algorithm for scheduling instruction execution.

Processor verification has been an active research topic in the recent years. *Model checking* [6, 7] has been one of the more successful techniques in hardware verification. An advantage of model checking is that it is completely automatic. However, straightforward application of it can not handle the complexity of modern processors because of the state explosion problem. It is also limited to verification of finite state systems, and can only prove correctness of a particular processor configuration.

McMillan [20] has used *compositional model checking* to verify a variant of Tomasulo's algorithm (see Section 2). The main idea in his paper is to use compositional reasoning to reduce the complexity of the verification. However, his technique is no longer fully automatic, and the user has to find a good balance of lemmas that, in effect, state the relatively complex invariant for induction over time.

Theorem proving can potentially handle a complex mix of data and control in a hardware device and is able to prove properties in general for arbitrary processor configurations. But direct theorem proving [10, 22] usually involves significant manual effort. Moreover, when a circuit has a complex control structure, a theorem prover has to consider a lot of cases arising from the branches in the control logic. This quickly makes the proofs too large and tedious to be manageable for real processors. Model checkers usually do not have this problem, since symbolic, or BBD-based, model checking techniques [19] are specifically designed to handle extensive case analysis.

An alternative approach based on theorem proving uses *completion functions* [14]. A completion function is a projection provided by the user that maps any OOO processor state into a *flushed state*, which corresponds to an *architectural state* of the machine visible by a sequential program executed on that processor. A simple way of reaching this state is by *flushing* the OOO machine directly, that is, running it without dispatching new instructions until all pending instructions are completed. If the completion function is simpler to compute than directly flushing the machine, then proofs can also be simplified. For Tomasulo's algorithm, direct construction of such a completion function was previously infeasible. A suitable completion function for an OOO processor with reorder buffer has been proposed in a recent paper [15]. This function is recursive, and the complexity is handled by performing induction on its recursion depth. Although feasible, this still requires significant manual effort and human insight to carry this induction over.

Burch and Dill [5] use the notion of *uninterpreted functions* to represent data and operations on data symbolically. The behavior of a processor is specified in terms of uninterpreted function symbols, and *symbolic execution* is performed. The result of each operation in such a representation is a term constructed from uninterpreted function symbols. Providing a concrete interpretation for these function symbols results in a particular run of a concrete processor being verified. All of the formulas with uninterpreted function symbols that can be proven to remain true under arbitrary interpretations. Thus, the approach of Burch and Dill can be used prove the correctness of a device without knowing the concrete implementation details.

Symbolic execution is based on extensive term rewriting and simple proof-theoretic reasoning, and thus can be easily automated. However, it requires special decision procedures

for uninterpreted function symbols and does not use previously existing techniques like BDDs [4]. Also the rate at which formulas grow with each cycle of the symbolic execution depends on the size of the circuit. Therefore, when the circuit is relatively large, the formulas quickly become unmanageable. In addition, the number of symbolic execution cycles required to complete the verification grows with the circuit size, making the approach non-scalable.

Skakkebæk et al. [23] propose an *incremental flushing* technique for verification of an OOO design. The technique is designed to overcome the fast growth of formulas in symbolic execution at the expense of automation, and heavily relies on human-guided theorem proving. Their approach is also based on uninterpreted function symbols and uses the SVC tool as a decision procedure.

Sajid et al. [23] have extended the decision procedures for uninterpreted function symbols to use BDDs. However, their work shares the disadvantage of [5]; the decision procedure they give can not be easily combined with symbolic model checking. They have also not investigated how their techniques can be applied to the verification of OOO processors.

Hojati and Brayton [13] have developed a formal description technique for integer combinational/sequential (ICS) systems. Their technique is even more general than uninterpreted functions ([5]). For a restricted class of models they show, by applying the notion of data independence [26], that 0–1 instantiation can reduce the size of the model to a finite number of states. This allows the usage of efficient symbolic model checkers. However, OOO execution is inherently data dependent and thus these abstraction techniques can not be applied. Hojati and Brayton also investigate how approximate reachability analysis can be performed for general ICS models. They give an algorithm that makes use of BDDs, but their technique is only used for reachability analysis and no limit on the number of terms occurring in the verification can be given.

Velev, Bryant, and German [3, 24] have developed methods of combining uninterpreted function symbols with BDD-based symbolic model checking. However, their work is mostly focused on completely automatic techniques, and can handle only fixed-size processor designs.

In this paper we introduce two main techniques: a new method of incorporating uninterpreted function symbols into traditional model checking, and a "light-weight" propositional completion function for Tomasulo's algorithm. This allows us to verify Tomasulo's algorithm in arbitrary an configuration; that is, for any number of registers, reservation stations, functional units, and an arbitrary instruction set.[1] We observe that the terms that appear during symbolic execution are not arbitrary. Often, they share subterms and have similar structure. We exploit this observation by introducing a special state representation that reduces the number of copies of identical subterms. This representation is based on a data structure called the *reference file*. Terms that share common subexpressions simply have references to the same entries in the reference file. This greatly reduces the number of bits needed for the state representation, thus reduces the number of states. It also simplifies the problem of checking equivalence between terms—we simply compare the references. Although structure sharing has been widely used for term representation in software, this is the first time the technique is used in a finite-state SMV program, and it is one of the main contributions of this paper. The use of reference file enables us to use traditional CTL model

checking for uninterpreted function symbols. This representation alone makes it possible to verify a Tomasulo's machine with up to 5 reservation stations [2].

In addition, the reference file representation allows us to construct a relatively simple propositional completion function for Tomasulo's algorithm. This dramatically reduces the complexity of verification, and since the specifications are propositional formulas, we can take advantage of SAT procedures to speed up model checking [2]. No complicated induction as in [15] is necessary for our completion function. To enhance the verification further, we combine it with theorem proving and compositional model checking, where the most tedious parts of the proofs are carried out by a model checking engine. The resulting technique permits a high degree of automation with low computational complexity, and at the same time is able to prove the correctness of Tomasulo's algorithm in its full generality. We believe that this new methodology can be extended to handle even more complex and realistic superscalar processor designs in the future.

Our paper is organized as follows. We describe the Out-Of-Order engine in Section 2. The completion function technique is explained in Section 3. Our encoding of the model is given in Section 4. Section 5 provides the details of the case splitting technique, and the entire verification process is summarized in Section 6. We give experimental results in Section 7 and conclude in Section 8.

## 2. Out-of-order execution

As a benchmark for our method we have verified an implementation of Tomasulo's *Out Of Order* (OOO) execution algorithm [12] in its general case. This algorithm is the basic technique that is used to implement OOO execution in modern microprocessors. It also has been used in previous work on the verification of OOO devices [10, 20]. We will explain the idea behind OOO execution and how Tomasulo's algorithm implements it. At the end of this section we briefly describe our model.

To achieve greater throughput of instructions, superscalar microprocessors use several functional units that can execute instructions in parallel. However, if two instructions depend on each other (e.g. the later needs the result of an earlier instruction) one of them has to wait until the other has finished. In this case, one functional unit is idle. But if a different instruction, potentially following the other two in the instruction sequence, does not depend on their results, then it can be executed in parallel on the free functional unit.

For instance, in figure 1 instruction I1 depends on the result computed by I0, and I2 does not have to wait for I0 or I1. This allows I2 to be executed in parallel with I0. Since a multiplication can take much longer than an addition, the execution of I2 could be finished before I1 has started. In this case R1 may be updated with the result of I2 before I1 starts execution, and the execution of I1 may read the wrong value from register R1. This

| | |
|---|---|
| I0 | R0 := R0 * R1 |
| I1 | R0 := R0 * R1 |
| I2 | R1 := R1 + R1 |

*Figure 1.* An instruction sequence that allows OOO execution.

| Name | Description | Example |
|---|---|---|
| RAW | *Read After Write*: instruction reading from a register may complete before the previous instruction writes to the same register. Since we need the result of the previous instruction, this situation is called the *true data dependency*. This hazard can only be prevented by waiting for the result of the previous instruction. | $I_1: \quad R_2 \times R_0 \to \mathbf{R_1}$<br>$I_2: \quad \mathbf{R_1} + R_2 \to R_3$ |
| WAR | *Write After Read*: instruction writing to a register may complete before the previous instruction reads from the same register. Can be prevented by saving the old value of the register in a separate location. | $I_1: \quad \mathbf{R_0} \times R_1 \to R_0$<br>$I_2: \quad R_1 + R_1 \to \mathbf{R_1}$ |
| WAW | *Write After Write*: two instructions writing to the same register may complete out of order, leaving the wrong value in the register. Can be prevented by either in-order dispatching and tagging the destination registers, or in-order completion. | $I_1: \quad R_4 \times R_5 \to \mathbf{R_1}$<br>$I_2: \quad R_2 + R_0 \to \mathbf{R_1}$ |

*Figure 2*. Definition of data hazards in out-of-order processor. The names of the hazards are derived from the intended order of events, and it is the hazard when this order can be violated.

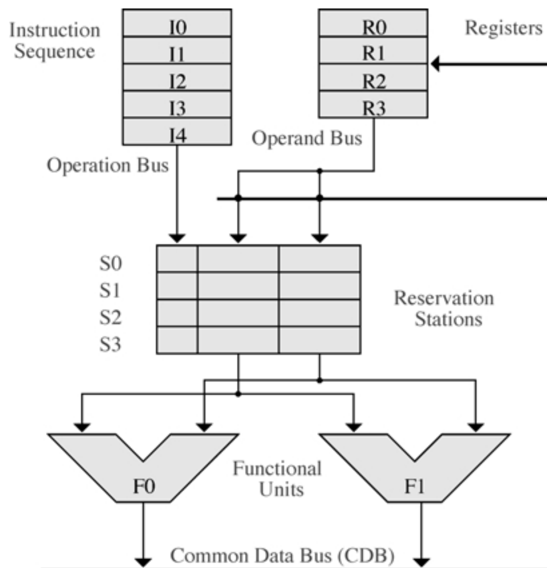


*Figure 3*. A model of an implementation of Tomasulo's algorithm.

is a *data hazard* ("write after read" in this case), and it must be handled properly. In general, a data hazard arises when changing the instruction execution order influences the result of the computation (see figure 2). Tomasulo's algorithm was designed to avoid such problems.

The main idea behind Tomasulo's algorithm is to *dispatch* instructions from the instruction sequence into a pool of *reservation stations* (figure 3). The arguments for each instruction are read from its source registers and placed into its reservation station. Then the destination register is marked with a special *tag* indicating that its value is not yet available

and will be produced by the instruction in this reservation station. Note, that previously issued instructions may have had the current source registers as their destination registers. So, the current source registers may already be tagged, in which case the tags are copied into the reservation station instead of the values.

When an instruction completes, its result is posted on the *Common Data Bus* (CDB) together with the reservation station number that is responsible for the instruction. Each tagged register compares its tag with the reservation station number on the bus, and if it matches, copies the value from the bus and clears its own tag. This mechanism is used to update both the register file (architectural registers), and the argument registers in the reservation stations.

From a reservation station an instruction is scheduled for execution on an unoccupied functional unit as soon as all its operands are available, i.e., when all tags are cleared and the corresponding data arrived. *Read after write* hazards, when the data must be computed before it can be used, i.e., the "true" dependences, are now handled by delaying the execution of an instruction until all of its operands are computed. *Write after read* hazards, when a new value may overwrite the old one that we need, are eliminated by copying the values for the dispatched instruction's arguments into the reservation station from the register file. Thus, no further updates to the source registers can overwrite the arguments. *Write after write* hazards (after executing two instructions writing to the same destination register, the value of the register must be the result of the second instruction) are resolved by in-order dispatching and tagging. Since the destination register is tagged no matter whether it has been tagged before, only the last tag will remain and cause the register to fetch the correct result from the bus. Figure 2 summarizes these hazards in a table.

A pictorial diagram of our model is given in figure 3. It is similar to the OOO unit described by [12] and the same as described by [2]. It consists of a set of registers (the *register file*), a pool of reservation stations, and several functional units. The important feature of our model is that reservation stations are not associated with specific functional units. We use a common pool of reservation stations because it makes the model more general and easier to verify. A similar model was independently used by McMillan [20]. This is not a completely artificial model: the Pentium Pro™ [11] microprocessor also has a common pool of reservation stations. We can model processors with reservation stations assigned to functional units simply by restricting the scheduling algorithm. Therefore, our method can be used, for example, for a PowerPC 604 processor.

## 3. The verification technique

Intuitively, the OOO processor is correct, if the result of any program executed on it is consistent with its sequential semantics. That is, the result must be the same if we execute this program on a sequential machine, which we call a *specification machine*. In our model, for any finite sequence of instructions, if the initial contents of the register file in both OOO and sequential specification machines are the same, and the OOO machine is "flushed" (that is, all reservation stations are empty, and all pending instructions in them are completed), then, when they both finish execution of that instruction sequence, the register files will again have the same values.

Formally, define $\mathsf{clock}(s)$ to be a state of the machine after executing it exactly one clock cycle from the state *s without dispatching any new instruction*. Note that clock also has implicit arguments besides the explicit argument $s$. These implicit arguments drive the OOO machine's internal scheduling decisions at each state. Such arguments determine to which reservation station a new instruction must be dispatched, and which instructions must be executed next on which functional units, if there is such a choice. We hide these arguments to simplify the notation, and because they are inessential for the verification purposes. The hidden arguments are only used by two *oracle* functions $\mathsf{complete}(s)$ and $\mathsf{scheduler}(s)$ which implement the details of internal scheduling in the OOO implementation. The benefit of using oracles is that a certain implementation of oracles immediately gives us a particular implementation of Tomasulo's algorithm. This way we can easier match our model with the actual designs. For the purpose of verification, we replace the oracles with nondeterministic choice, which covers all possible scheduling choices, thus verifying the machine in the most general setting for all possible scheduling algorithms. We will define all these functions more precisely later in this section.

*Definition 3.1.* An OOO state $s$ is called flushed iff all reservation stations in $s$ are empty (contain no pending instructions).

An OOO state $\mathsf{flush}(s)$ is the result of *flushing* the machine from the state $s$; it is equivalent to executing the machine until all of the instructions in the reservation stations are completed, that is $\mathsf{flush}(s) = \mathsf{clock}^n(s)$ for some $n \geq 0$. The function $\mathsf{dispatch}(s, i)$ yields the state after dispatching a new instruction $i$ in the machine in a state $s$. Note that $\mathsf{dispatch}(s, i)$ also runs the machine for one clock cycle, which may let another instruction retire (that is, complete execution and write back the results). In our model we allow at most one instruction to be dispatched and at most one to retire in the same clock cycle. Function $\mathsf{exec}(s, i)$ returns a state of the machine after executing an instruction $i$ in a *flushed* state $s$, as specified by the sequential semantics. That is, the instruction $i$ is executed with the current values of the registers and the result is immediately written back to the register file.

Note that flush and dispatch are partial functions, since not every state can be flushed (e.g. due to possible cyclic dependencies among reservation stations), and not every state has a reservation station available for dispatching a new instruction. We will define all these functions explicitly in Section 4.2.

*Definition 3.2.* For any instruction sequence $\pi = i_0, \ldots, i_{n-1}$ of length $n \geq 0$ define the set of OOO traces $\mathsf{tr}_{ooo}(s_0, \pi)$ to be the set of sequences of states $s_0, \ldots, s_n$ such that the initial state $s_0$ is flushed and $s_{j+1} = \mathsf{clock}^{k_j}(\mathsf{dispatch}(s_j, i_j))$, where $k_j \geq 0$ is such that $s_{j+1}$ has a free reservation station and another instruction can be dispatched to it in the next cycle. That is, $s_j$ is some state of the machine after dispatching all the first $j$ instructions from $\pi$ in the originally empty machine, and, in addition, the machine in the state $s_j$ is ready to accept a new instruction.

Note that $\mathsf{tr}_{ooo}(s_0, \pi)$ may contain more than one sequence, each corresponding to different values of the hidden decision arguments of clock. It may also be empty if the processor

has an error and can get into a deadlock state—for instance, due to cyclic dependencies among reservation stations.

*Definition 3.3.* For any instruction sequence $\pi = i_0, \ldots, i_{n-1}$ of length $n \geq 0$ define the set of sequential traces $\mathsf{tr}_{seq}(q_0, \pi)$ to be the set of sequences of states $q_0, \ldots, q_n$ such that the initial state $q_0$ is flushed, and $q_{j+1} = \mathsf{exec}(q_j, i_j)$.

Since the sequential execution is deterministic, $\mathsf{tr}_{seq}(s_0, \pi)$ is always a singleton set, and can be defined as a function returning the trace. But we define it as a set to be consistent with $\mathsf{tr}_{ooo}$.

We denote a $j$-th state of each trace by $s_j$ for $\vec{s} \in \mathsf{tr}_{ooo}(s_0, \pi)$ and $q_j$ for $\vec{q} \in \mathsf{tr}_{seq}(s_0, \pi)$ respectively, where $\vec{s}$ and $\vec{q}$ denote the entire traces. Note that the initial state $q_0$ is always the same as $s_0$. Now the main correctness property of the OOO algorithm can be stated as the following theorem:

**Theorem 3.4.** *For an arbitrary flushed state $s_0$, an instruction sequence $\pi$ of length $n$, and any traces $\vec{s} \in \mathsf{tr}_{ooo}(s_0, \pi)$ and $\vec{q} \in \mathsf{tr}_{seq}(s_0, \pi)$ the following holds*:

$$\mathsf{flush}(s_n) = q_n.$$

Here $s_n$ is the OOO state of the machine after dispatching all instructions from $\pi$, and in the $\mathsf{flush}(s_n)$ state all these instructions must be completed. At this point, the state of the machine must be exactly the same as $q_n$, the state after executing the same instructions sequencially on the specification machine (figure 4). This theorem can be vacuously true for a certain initial state $s_0$ and an instruction sequence $\pi$ if the machine cannot be flushed (e.g. the processor gets to a deadlock state due to a design error), and the set of OOO traces $\mathsf{tr}_{ooo}(s_0, \pi)$ is empty. Therefore, the theorem only states the *safety property*, that the OOO machine never does anything wrong if it does anything at all. The ability to flush the machine from any reachable state is a *liveness property*, and could also be handled in our model. However, we do not consider it in this paper, and concentrate only on the safety property.

The proof of Theorem 3.4 is done by induction over the instruction sequence length (figure 4). The base of the induction ($n = 0$) is trivial, since $\mathsf{flush}(s_0) = s_0 = q_0$ by definition. The inductive step states that for any reachable state $s_j$, flushing the machine and executing an instruction $i_j$ results in the same state as when we first dispatch this instruction and then flush the machine (figure 4):

$$\forall j \geq 0.\ q_j = \mathsf{flush}(s_j) \Longrightarrow q_{j+1} = \mathsf{flush}(s_{j+1}),$$

where $q_{j+1} = \mathsf{exec}(q_j, i_j)$. We call this property the *Burch and Dill Commutative Diagram* [5]. If we express $s_{j+1}$, $q_j$, and $q_{j+1}$ in terms of $s_j$:

$$s_{j+1} = \mathsf{clock}^{k_j}(\mathsf{dispatch}(s_j, i_j)), \quad q_j = \mathsf{flush}(s_j),$$

where $k_j \geq 0$ is the number of cycles to run the machine after dispatching $i_j$ before there is a free reservation station (see Definition 3.2). By Definition 3.3, we also have:

$$q_{j+1} = \mathsf{exec}(q_j, i_j) = \mathsf{exec}(\mathsf{flush}(s_j), i_j),$$

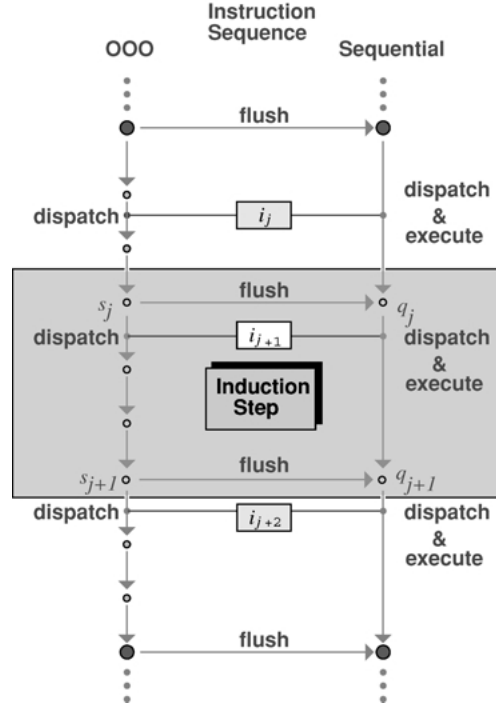


*Figure 4.* Induction over the instruction length.

and notice that

$$\text{flush}(s_{j+1}) = \text{flush}(\text{clock}^{k_j}(\text{dispatch}(s_j,\ i_j))) = \text{flush}(\text{dispatch}(s_j,\ i_j)),$$

since running a machine for a few cycles and then flushing is the same as just flushing it right away. From the above, we obtain a "commutative" property of flushing and dispatching/executing that we need to show in order to prove the inductive step:

$$\forall j \geq 0.\ \text{exec}(\text{flush}(s_j), i_j) = \text{flush}(\text{dispatch}(s_j, i_j)).$$

As in any induction, this diagram may not hold in general for arbitrary states $s_j$, and we need an *inductive invariant* that restricts this property to some superset of *reachable states* where it holds. We call a state $s$ *reachable* if the machine can reach this state by executing some instruction sequence from some initial state. Formally, $s$ is reachable if there is an initial (flushed) state $s_0$ and an instruction sequence of some length $n$ such that $s = s_n$, where $s_n$ is defined as in Theorem 3.4. A sufficient inductive invariant $I$ in our case happens to be relatively simple:

- $I(s)$: Tags both in registers and reservation stations in state $s$ point only to reservation stations that have instructions to be executed in them.

Since no tags are in use in the initial state $s_0$ (because it is flushed), $I(s_0)$ is vacuously true in the initial state. The commutative diagram now is a corollary of the following lemma:

**Lemma 3.5.** $\forall s.\ \forall i.\ I(s)$ *holds and*

$$\mathsf{flush}(s),\ s' = \mathsf{dispatch}(s,i),\ \textit{and}\ \mathsf{flush}(s')\ \textit{are all defined}$$
$$\textit{implies}\ I(s') \wedge \mathsf{exec}(\mathsf{flush}(s),i) = \mathsf{flush}(s').$$

Additionally, we need to ensure the invariant $I(s)$ holds not only right after dispatching the instruction $i$, but also at some future state $\mathsf{clock}^k(s')$ in which the machine is ready to dispatch the next instruction. For this it is sufficient to prove the following lemma:

**Lemma 3.6.**

$$\forall s.\ I(s) \Longrightarrow I(\mathsf{clock}(s)).$$

Notice that proving the commutativity of the diagram (Lemma 3.5) requires flushing the OOO machine twice. Since we are effectively using symbolic execution whose complexity increases exponentially with the number of cycles of the OOO machine, flushing it may be very costly. Thus, we would like to do an inferior induction over the number of clock cycles required to flush the machine to ease the burden of the model checker. This inferior induction will inevitably require another inductive invariant, perhaps much stronger than the one we already have. The work of Jens Skakkebæk et al. [23] suggests that such an invariant, when expressed directly, is quite complicated. Moreover, it requires significant changes to the model, special treatment of nondeterminism, and sometimes even intermediate abstraction layers.

### *3.1. Completion function approach*

There is, however, a different way to express the same invariant using a *completion function* [14]. A completion function is a function $f$ that takes an OOO state $s$ and returns a sequential state $f(s)$ corresponding to the flushed state of the machine from $s$ if one exists. If the machine cannot be flushed from $s$ (e.g. due to cyclic dependencies among reservation stations), the result of $f(s)$ can be any state. The hard way to compute such a function would be to run the OOO machine from a state $s$ without dispatching new instructions until it is flushed. However, if the user can provide a completion function $f$ with lower complexity, then our commutative diagram (Lemma 3.5) becomes

**Lemma 3.7.** $\forall s.\ \forall i.\ I(s) \Longrightarrow I(\mathsf{dispatch(s,i)}) \wedge \mathsf{exec}(f(s),i) = f(\mathsf{dispatch}(s,i)).$

The proof of this lemma becomes much easier than the one for Lemma 3.5 with direct flushing if the function $f$ is much simpler to compute than to flush the machine directly. However, since the function $f$ can still be a complex expression, we need to prove that it is indeed a completion function. Formally, this property can be expressed as a lemma:

**Lemma 3.8.** *If, for a state $s$, there is an $n \geq 0$ such that $\mathsf{clock}^n(s)$ is flushed, then $f(s) = \mathsf{clock}^n(s)$, where* clock *performs exactly one step of execution of the OOO machine without dispatching a new instruction.*

We prove this lemma by induction over the number of cycles required to flush the OOO machine. Suppose we are in an OOO state $s$. The important property is that $f$ always returns a flushed state, and if the original state $s$ is flushed, then the result of $f(s)$ must be the same as $s$.

The *base case* is when $s$ is already flushed ($n = 0$ in Lemma 3.8). Then we need to check that

$$f(s) = s.$$

The *inductive step* assumes that Lemma 3.8 holds for any $n - 1 \geq k > 0$:

$$\forall s'.\ \mathsf{clock}^k(s') \text{ is flushed} \Longrightarrow f(s') = \mathsf{clock}^k(s').$$

To prove the lemma for $k = n$ we instantiate $s'$ in the induction hypothesis by $\mathsf{clock}(s)$, obtaining the following:

$$\begin{aligned} &\forall s.\ \mathsf{clock}^n(s) \text{ is flushed} \Longrightarrow \\ f(\mathsf{clock}(s)) &= \mathsf{clock}^{n-1}(\mathsf{clock}(s)) \quad \text{(by ind. hypothesis)} \\ &= \mathsf{clock}^n(s), \end{aligned}$$

To complete the inductive step, we have to show that

$$\forall s.\ f(\mathsf{clock}(s)) = f(s).$$

Intuitively, flushing the machine immediately is the same as running it for one clock cycle and then flushing, and the completion function $f$ has to satisfy this property. In general, this property may not hold as stated for an arbitrary state $s$, since there are "invalid" states which cannot be flushed (e.g. due to a tag pointing to an empty reservation station), and the behavior of the completion function is not specified for such states. Hence, the property needs to be strengthened. It turns out that the same invariant $I$ that we mentioned earlier is sufficient for this lemma to go through. Thus, the actual sublemma for the inductive step is the following:

**Lemma 3.9.** $\forall s.\ I(s) \Longrightarrow I(\mathsf{clock}(s)) \wedge f(\mathsf{clock}(s)) = f(s).$

Note, that this lemma also implies Lemma 3.6. The reason that our inductive invariant is so simple compared to [23] is because all of the complexity is now implicitly encoded into the completion function.

In the remaining sections we give our implementation of Tomasulo's algorithm, its encoding using a special data structure called a *reference file*, and define our completion function. Then we apply another reduction technique known as *universal skolemization* in theorem proving, or *case splitting* in the model checking community. This generates a

number of smaller lemmas to prove. The number of lemmas is further reduced by symmetry reductions, and it makes it possible to prove the main correctness property (Theorem 3.4) for an arbitrary configuration of our Tomasulo's algorithm model.

## 4. Encoding of Tomasulo's algorithm

In order to verify Tomasulo's algorithm we need to abstract the concrete details of data manipulation in the functional units. We reformulate the algorithm using *uninterpreted functions* in place of concrete ALU operations in the same way as Burch and Dill [5]. For simplicity we focus only on arithmetic instructions that take two arguments and produce one result. The arguments are read from source registers, and the result is written back to its destination register. Each instruction performs a distinct operation $f_{op}$ specified by its *opcode* $op \in \mathsf{Op}$. If the source registers contain values $x$ and $y$, then the instruction produces the result $f_{op}(x, y)$. In our framework, the function $f_{op}$ is uninterpreted, and thus, the result $f_{op}(x, y)$ is not really computed, but is rather treated and recorded in the destination register as a *term*.

We start with the same basic idea as Burch and Dill [5]. Our model of the microprocessor does not compute concrete values. It only manipulates symbolic terms made of constants and uninterpreted function symbols. This is explained in figure 5(a) where a symbolic execution trace of a sequential microprocessor is shown. This example processor has two registers and an instruction sequence consisting of two instructions. The registers R0 and R1 contain the initial symbolic values r0 and r1 respectively. The first instruction adds these two values and stores the symbolic result '$\mathrm{r0} + \mathrm{r1}$' into register R0. Note that '$+$' is treated as an uninterpreted function with no actual meaning. In particular, we can not assume commutativity or associativity of these functions. The second instruction computes

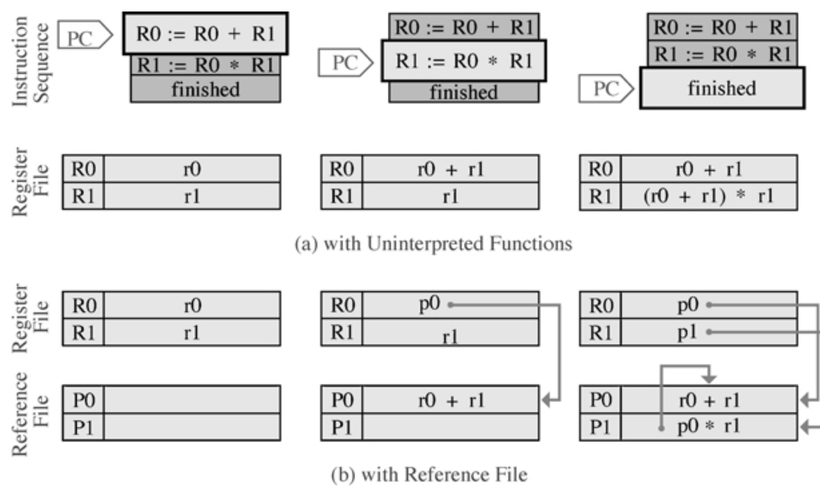


*Figure 5.* Execution trace of a sequential machine with and without reference file.

the product of the result and the initial value r1 of register R1. The final symbolic result '(r0 + r1) * r1' is generated and stored in register R1.

In general, we associate different uninterpreted constants with the initial values of the registers. Since the number of registers is finite, the number of constants is also finite. Thus, if we execute a finite number of instructions, the number of terms that occur during this symbolic execution will also be finite. Consequently, the processor model will be finite and can, in principle, be represented in a finite state model checker. A *direct encoding* of all possible terms would allow us to use model checking for the model with uninterpreted function symbols. However, we will show that the number of bits needed to represent these terms grows exponentially with the number of instructions. This makes this approach infeasible.

In general, any term occurring in the symbolic execution will have a height no larger than $i + 1$, where $i$ is the number of instructions. The height of a term is defined as the height of its syntax tree or, equivalently, the maximal number of nested function applications plus one. As a rough lower bound on the number of all possible terms we use a lower bound on the number of terms with maximal height and maximal number of subterms. Since all function symbols are binary in our example, the number of atomic terms in the largest terms is $2^i$. In each place there can be one of $r$ constants, thus, the total number of different maximal terms is $r^{2^i}$.

Since this is doubly exponential, the number of bits needed in a binary encoding of that domain would grow exponentially with the number of instructions. As an example consider the case where $r = 4$ and $i = 5$. In a binary encoding we have to use at least $\log_2 4^{2^5} = 2^6 = 64$ bits for each register and other locations where a data value can be stored. With four registers and four reservation stations the number of state bits would be at least $64 \cdot (4 + 2 \cdot 4) = 768$. Note that 64 bits is often as big as the width of a register in a concrete model.

### 4.1. *The reference file*

While the brute force approach of direct encoding of all possible terms is not feasible, it is important to note that in one execution trace not all possible terms can occur. Moreover, the same terms or subterms are *referenced* at different locations. For instance, in the final state of the execution trace in figure 5(a) the subterm 'r0 + r1' occurs both in register R0 and R1. In this model it has to be stored twice and can not be shared.

A similar problem occurs in the implementation of logic and functional programming languages like Prolog and Lisp [17, 18, 25]. They use a *heap* to store newly generated terms. Registers in the abstract machine for these languages (e.g. WAM [25]) only contain constant values or pointers to the heap. This prevents unnecessary copying and allows sharing of common subterms.

We use a special data structure, called a *reference file*, similar to a heap. This is a much more compact encoding of the terms produced during an execution. Each entry of the reference file contains an application of an uninterpreted function symbol. Each operand of the function application is either an initial value of a register or a pointer to another entry of the reference file. Unlike the heap, the size of the reference file is finite and is equal to the number of instructions $i$.

As an example, figure 5(b) shows the execution of the same instruction sequence as in figure 5(a). Now all terms are represented with a reference file. After the first instruction the entry P0 of the reference file stores the corresponding function symbol ('+') together with its operands. In our case the operands are the constants 'r0' and 'r1'. Instead of the whole term ('r0 + r1') only the pointer ('p0') is written into the destination register R0. This result is further used by the second instruction. In the entry P1 of the reference file allocated for this instruction the first operand is again stored as a pointer ('p0') without being expanded. Finally, the pointer 'p1' is written to the destination register R1 of the second instruction.

Compared to the first execution trace in figure 5(a), the difference is that the registers do not contain terms anymore. Instead, symbolic constants ('r0' and 'r1') or pointers to the reference file ('p0' and 'p1') are stored in the registers. When a functional unit finishes a computation, a new entry is allocated in the reference file for a newly generated term, and the registers are updated with pointers to it. Note that the terms occurring in the first run can easily be restored from the reference file by expanding the pointers.

Since in an OOO machine the reference file may be filled in different order than in a corresponding sequential machine run, we need to make sure we can still compare reference file contents in the two machines at the end. One way to do this is to index the reference file by the opcode. That is, when a new term is created by executing an instruction with an opcode *op*, it is placed in the *op*'s slot of the reference file. If all the opcodes in the instruction stream are different, then the order of terms in the reference file does not depend on the execution order. Moreover, we do not have to record the opcode in the reference file for a new term, since it is now implicitly encoded into the index. The reference file entry therefore stores only the two operands of the instruction $\mathsf{arg}_1$ and $\mathsf{arg}_2$, and if they are stored in an entry $p$, then it represents the term $f_p(\mathsf{arg}_1, \mathsf{arg}_2)$.

The requirement that all the opcodes should be different does not compromise the generality of the verification, since the opcodes are just uninterpreted symbols. If later we want to instantiate such a symbolic sequence of instructions with an actual program that has repeated instructions, we simply interpret several symbolic opcodes with the same concrete value.

This assumption brings an additional *inductive invariant* that all opcodes differ at all times, which, in particular, requires the newly dispatched instruction to have an opcode not used before.

Now we can calculate an upper bound on the number of bits for the representation of the reference file. Each entry has to store two operands, where an operand is a constant or a pointer to the reference file. For $i$ instructions and $r$ registers there are $i + r$ values for each operand. This requires $2 \cdot \log_2(i + r)$ bits to encode the two operands per entry. The entire reference file consists of $i$ entries, and thus, can be encoded with $O(i \cdot \log_2(i + r))$ bits.

We also need to encode the contents of the register file and the reservation stations. There, in addition to data values, we also need to store tags (see Section 2). For $t$ reservation stations this requires $\log_2(i + r + t)$ bits, since we have $t$ different tag values. Each reservation station contains a busy bit, two operands and an opcode. This takes $t \cdot (1 + 2 \cdot \log_2(i + r + t) + \log_2 i)$ bits for all $t$ reservation stations. We also need $r \cdot \log_2(i + r + t)$ bits for $r$ registers. Putting it all together, our model of Tomasulo's algorithm can be encoded with $O((i + r + t) \cdot \log_2(i + r + t))$ state bits.

For an example with four registers ($r = 4$), four reservation stations ($t = 4$), and five instructions ($i = 5$) the exact number of bits is 16 for the register file, 48 bits for the reservation stations, and 55 bits for the reference file, giving a total of 119 bits.

### 4.2. *Completion function with reference file*

With a direct representation, any naively generated completion function for Tomasulo's algorithm would be nearly as hard to compute as just flushing the machine step by step. The reason is that the final value of a register may, in the worst case, depend on all the reservation stations, when every instruction needs the result of the previous one. Thus, simply tracing the datapath, recording the transformations made to the data, and rewriting it as a function will yield an expression involving the entire system, even if it returns only the value of one register. Thus, the verification remains as complex as the brute force model checking of the entire system and is not scalable.

The main reason that the complexity of the completion function is so high when written in a straightforward way is that it mentions all or almost all of the components of the system. If we can write a completion function that only references a fixed number of components for each register, regardless of the actual size of the design, then the complexity will be bounded for each register, and, therefore, grow linearly with the number of registers. It turns out that using our reference file representation we can write the completion function in such a scalable way.

We denote the entire OOO state by the 4-tuple:

$$s = (R, \mathsf{tag}, \mathsf{rff}, \mathsf{rs}),$$

and the result of applying the completion function to it by $f(s) = (R', \emptyset, \mathsf{rff'}, \emptyset)$, where $R$ is the register file, tag is the array of tags for the registers, rff is the reference file, and rs is the pool of reservation stations. The register file $R$ is the array of register values, and does not include the tag part. When a register is not marked by a tag, the corresponding tag entry has a special value $\emptyset$. The reference file rff is the array of value pairs. Recall that each reference file entry needs to hold only the values of the two arguments. The opcode of the instruction is encoded implicitly into the index of that entry. Each reservation station in the array rs is a tuple $(\mathsf{op}, \mathsf{arg}_1, \mathsf{tag}_1, \mathsf{arg}_2, \mathsf{tag}_2)$ holding the opcode and the two arguments with their tags.

The components of the state are extracted by their names. For instance, the value of the entire register file in the state $s$ is $R(s)$, and an individual register $r$ is $R_r(s)$. Similarly, the the first argument of the instruction from reservation station $t$ is $\mathsf{arg}_1\ (\mathsf{rs}_t(s))$, and its tag is $\mathsf{tag}_1\ (\mathsf{rs}_t(s))$.

The completion function $s' = f(s)$ is defined below by the components $R(s')$ and $\mathsf{rff}(s')$. The tags and reservation stations are empty after flushing the machine, and we use $\emptyset$ to reflect this.

Recall that the reference file is indexed by the opcodes, and every time we construct a term with an opcode $p$, we place it into the corresponding reference file entry and refer to it by a pointer—that is, the opcode $p$ itself.

For each register, if its value is not ready and its tag is pointing to a reservation station holding an opcode $p$, then eventually this register will hold the pointer $p$. Otherwise its value remains the same (we use some uninterpreted constants to represent initial values of the registers). Notice that we reference only one reservation station per register. The pseudo-code for this part of the completion function is

For every register $r$
  **if** $\mathsf{tag}_r(s) \neq \emptyset$ **then** $R_r(s') := \mathsf{op}(\mathsf{rs}_{\mathsf{tag}_r}(s))$;
  **else** $R_r(s') := R_r(s)$;

Here $R_r(s)$ is the register numbered $r$, and $\mathsf{tag}_r$ is its tag which could be *inactive* (=∅) if the register is untagged. $R_r(s')$ is the value of the same register after flushing the machine (or, more precisely, after applying the completion function $f$). The op operator extracts the opcode from the reservation station pointed to by $\mathsf{tag}_r$.

Since we will use this tag expansion mechanism again for the reference file, we define it as a new function:

**function** expand($s$, val, tag)
  **if** tag $\neq \emptyset$ **then return** $\mathsf{op}(\mathsf{rs}_{\mathsf{tag}}(s))$;
  **else return** val;

And the final value of a register $r$ can now be computed as

$$R_r(s') := \mathsf{expand}(s, R_r(s), \mathsf{tag}_r(s)).$$

For each entry $p$ of the reference file we check whether there is a reservation station that holds this opcode. If there is one, then the value of this reference file entry will be a pair of the final values of the arguments in that reservation station. Recall that we do not store the entire term in the reference file entry that results from the execution of the instruction. Since the reference file is indexed by opcode, it is sufficient to store only the two arguments of a corresponding instruction in each entry (see Subsection 4.1).

Each argument in the reservation station is similar in its structure to a register (it has a "value" field and a tag), and its final value is computed in the same way as a register. That is, for each argument we look at its tag and either copy the "value" field or follow the tag to another reservation station and extract its opcode. In total, we have to look into at most 3 reservation stations to compute the final value of the reference file entry. In pseudo-code, this can be written as follows:

For every opcode $p$
  **if** ∃ reservation station $t$ such that $\mathsf{op}(\mathsf{rs}_t(s)) = p$ **then**
    $\mathsf{rff}_p(s') := (\mathsf{arg}'_1, \mathsf{arg}'_2)$;
    **where**
      $\mathsf{arg}_1(s') = \mathsf{expand}(s, \mathsf{arg}_1(\mathsf{rs}_t(s)), \mathsf{tag}_1(\mathsf{rs}_t(s)))$;
      $\mathsf{arg}_2(s') = \mathsf{expand}(s, \mathsf{arg}_2(\mathsf{rs}_t(s)), \mathsf{tag}_2(\mathsf{rs}_t(s)))$;
  **else** $\mathsf{rff}_p(s') := \emptyset$;

```
function f(s) =
  For every register r:
    R_r(s') := expand(s, R_r(s), tag_r(s));
  For every opcode p
    if ∃ reservation station t such that op(rs_t(s)) = p then
      rff_p(s') :=(arg'_1, arg'_2);
      where
        arg_1(s') = expand(s, arg_1(rs_t(s)), tag_1(rs_t(s)));
        arg_2(s') = expand(s, arg_2(rs_t(s)), tag_2(rs_t(s)));
    else rff_p(s') := ∅;
  return(s');
```

*Figure 6.* Completion function $f$ in pseudo-code.

```
function exec(s, i) =
  For every register r:
    if r = dest(i) then R_r(s') := op(i);
    else R_r(s') := R_r(s);
  For every reference file entry p:
    if p = op(i) then rff_p(s') :=(R_src1(i)(s), R_src2(i)(s));
    else rff_p(s') := rff_p(s);
  return s';
```

*Figure 7.* Pseudo-code for function exec.

Here $\mathsf{arg}_j(\mathsf{rs}_t(s))$ and $\mathsf{tag}_j(\mathsf{rs}_t(s))$, $j \in \{1, 2\}$, are the value field and the tag of the $j$-th argument in reservation station $t$. A reference file entry can be either *empty* (∅), if there is no instruction with the corresponding opcode, or it can be a pair of values representing the two arguments of the instruction. The entire completion function is summarized in figure 6, and can be easily expressed in a specification language like SMV.

Now we give pseudo-code for exec, clock, and dispatch. For exec we only need to define the values of registers and the reference file, since the other fields are not needed for sequential execution and do not change. $\mathsf{op}(i)$, $\mathsf{src}_1(i)$, $\mathsf{src}_2(i)$, $\mathsf{dest}(i)$ are the opcode, two source registers, and one destination register, respectively, of the instruction $i$ (figure 7).

Intuitively, we update only the destination register with the pointer to a reference file entry, and create a new term in that entry using the values in the source registers. All the other parts of the state remain the same.

The $\mathsf{clock}(s)$ "executes" the machine for one clock cycle from the state $s$ without dispatching new instructions. The processor either stays idle (the state does not change), or *completes* one instruction from some reservation station that has all of its arguments ready (i.e. arguments are *untagged*). Completion means writing back the value of the instruction to the registers and removing the instruction from its reservation station when the execution has finished. In our model we execute the instruction at the time of completion. We use a special oracle $\mathsf{complete}(s)$ that returns a reservation station containing an instruction with its arguments ready, or ∅, if no instruction should be completed in the current cycle. We do not require the machine to complete an instruction as soon as it is ready; this allows us to model multi-cycle execution time for some instructions.

```
function bus(s) =
  let t = complete(s) in
    if t = ∅ then tg' := ∅;
    else
      tg' := t;
      op' := op(rs_t(s));
      res' := (arg_1(rs_t(s)), arg_2(rs_t(s)));
    endif
  return (tg', op', res');
```

*Figure 8.* Pseudo-code for function bus.

First, we define an auxiliary function $\mathsf{bus}(s)$ that computes the value of the Common Data Bus (CDB) where the result of a completed instruction is written back. The bus has three fields: tg, op, and res that denote the reservation station number where the completed instruction is stored, the opcode of the instruction, and the term resulting from the execution of the instruction compatible with the reference file representation (that is, it is a pair of the arguments' values). If no instruction is completed in the current cycle, then the value of the tg field is $\emptyset$ (figure 8). Notice that the function bus constructs the term resulting from the instruction execution, which in a real processor is the job of the functional units. But since in our model we only generate terms and do not compute the real values, it is very easy to write the combination of functional units and the bus in one single function. It also makes our model simpler and easier to understand.

The clock function is defined in figure 9. Similarly to the completion function, the return value is composed of the primed variables. It fully defines the functionality of the OOO Tomasulo machine when no instructions are being dispatched, and therefore is relatively complex. First, we check whether any instruction has been completed, and if none, then the processor idles and the state remains the same. When an instruction completes from a reservation station $t$, we must free that reservation station (assign $\emptyset$ to it) after updating all the internal registers that are waiting for this result. A register is waiting for the result of the reservation station $t$ if its tag is pointing to $t$. For each register with the tag $t$ we erase the tag and write the opcode from the bus to that register (which is a pointer to the corresponding term in the reference file). Otherwise the values of both the register and the tag remain the same. This has to be done for both the architectural registers $R_r$ and the argument registers $\mathsf{arg}_1$ and $\mathsf{arg}_2$ in the non-empty reservation stations. The reference file entry for the completing instruction is also updated with the term on the bus.

We are ready to define the dispatch function (figure 10). The dispatch function updates the tag in the destination register and the reservation station where the new instruction is dispatched. It also allows the OOO machine to execute one clock cycle in parallel, and it uses the clock function to achieve that. The new instruction can be dispatched into any *free* reservation station (one that does not have an instruction in it) which it is determined by a *scheduler*. In our pseudo-code we use another oracle $\mathsf{scheduler}(s)$ that returns a free reservation station if there is one, or a special symbol $\emptyset$ otherwise. We also assume that we only apply dispatch to those states $s$ where $\mathsf{scheduler}(s) \neq \emptyset$, and do not include this check in the code.

```
function clock(s) =
  let t = tg(bus(s)) in
                      /* No write back, state doesn't change */
    if t = ∅ then return s;     else
                      /* Otherwise update all values in: */
                      /* Register File */
      For every register number r:
        if tag_r(s) = t then
          R_r(s') := op(bus(s));  /* Pointer to the ref. file entry */
          tag_r(s') := ∅;
        else
          R_r(s') := R_r(s);
          tag_r(s') := tag_r(s);
                      /* Reference File */
      For every reference file entry p:
        if p = op(bus(s)) then rff_p(s') := res(bus(s));
        else rff_p(s') := rff_p(s);
                      /* Reservation Stations */
      For every reservation station t1:
        if t1 = t then rs_t1 := ∅;
        elsif rs_t1 ≠ ∅ then
                      /* Argument 1 */
          if tag1(rs_t1(s)) = t then
            arg1(rs_t1(s')) := op(bus(s));  /* Pointer to the ref. file entry */
            tag1(rs_t1(s')) := ∅;
          else
            arg1(rs_t1(s')) := arg1(rs_t1(s));
            tag1(rs_t1(s')) := tag1(rs_t1(s));
                      /* Argument 2 */
          if tag2(rs_t1(s)) = t then
            arg2(rs_t1(s')) := op(bus(s));  /* Pointer to the ref. file entry */
            tag2(rs_t1(s')) := ∅;
          else
            arg2(rs_t1(s')) := arg2(rs_t1(s));
            tag2(rs_t1(s')) := tag2(rs_t1(s));
  return s';
```

*Figure 9.* Function $\mathsf{clock}(s)$ in pseudo-code.

The reference file is not influenced by the newly dispatched instruction, and can only change due to the completion of a previously issued instruction. The destination register must be tagged and the tag must point to the reservation station $t$ where the new instruction is dispatched. Then the reservation station $t$ must receive the opcode of the new instruction, and the arguments are read from the registers and copied to the reservation station. However, if a source register was tagged and at the same cycle the value for this tag appears on the bus, we must be able to detect this and forward the value from the bus instead of the register, which is done by the bypass function. If we do not forward the value, the tag that we copy from that register will point at the end of the cycle to a reservation station that has just completed, and, hence, is empty. This would violate our invariant $I$.

```
function dispatch(s, i) =
  let t = scheduler(s) in
                          /* Reference File: not directly affected */
    rff(s') := rff(clock(s));
                          /* Register File: update tag in destination */
    For every register number r:
      R_r(s') := R_r(clock(s));
      if r = dest(i) then tag_r(s') := t;
      else tag_r(s') := tag_r(clock(s));
                          /* Reservation stations */
    For every reservation station t_1:
      if t_1 ≠ t then rs_{t_1}(s') := rs_{t_1}(clock(s));
      else
        op(rs_{t_1}(s')) := op(i);
        (arg_1(rs_{t_1}(s')), tag_1(rs_{t_1}(s'))) := bypass(s, R_{src_1(i)}(s), tag_{src_1(i)}(s));
        (arg_2(rs_{t_1}(s')), tag_2(rs_{t_1}(s'))) := bypass(s, R_{src_2(i)}(s), tag_{src_2(i)}(s));
  return s';

                          /* Forwarding logic from the bus */
function bypass(s, val, tag) =
  if tag ≠ ∅ ∧ tag = tg(bus(s)) then
    val' := op(bus(s));  /* Pointer to the ref. file entry */
    tag' := ∅;
  else
    val' := val; tag' := tag;
  return (val', tag');
```

*Figure 10.* Function dispatch($s$, $i$) in pseudo-code.

We do not explicitly define the oracles complete and scheduler. In our SMV implementation they return nondeterministic values, and therefore, the correctness we prove extends to any scheduling algorithm in a concrete processor.

## 5. Case splitting transformation

So far we have discussed the high-level steps of the verification of Tomasulo's algorithm, which is also summarized in figure 12, in Section 6. Our primary goal is to show that the OOO machine is equivalent to the sequential (specification) machine for arbitrary instruction sequences. This is proven by induction over the instruction sequence. The base case, when the sequence is empty, is trivial. The induction step comprises the *Commutative Diagram* of Burch and Dill [5].

To prove the commutative diagram, we use the *Reference File encoding* of terms. This encoding allows us to write a *Completion Function* $f$ which eliminates the need for an expensive flushing operation. The commutative diagram property is then reformulated in terms of $f$. However, since $f$ is provided by the user, we also need to prove that $f$ is indeed a completion function, that is, it is equivalent to flushing. This is done by induction over the number of cycles required to flush the OOO machine. The base case is again simple,

and the induction step essentially shows that flushing the machine immediately using $f$ is the same as first running it for one clock cycle and then flushing it.

At the end of this stage, we are left with two properties: the commutative diagram in terms of $f$ in the reference file representation, and the induction step for the function $f$. These two properties are still hard to prove directly, since they heavily rely on the OOO model.

In this section, we describe the *Case Splitting* transformations to make the properties small enough for direct application of model checking.

Now, let us look back at the commutative diagram with the completion function (omitting the invariant for brevity):

$$\mathsf{exec}(f(s), i) = f(\mathsf{dispatch}(s, i)).$$

Separated by the individual components, the diagram can be written as:

$$\forall r.\ R_r(\mathsf{exec}(f(s), i)) = R_r(f(\mathsf{dispatch}(s, i)))$$
$$\forall j.\ \mathsf{rff}_j(\mathsf{exec}(f(s), i)) = \mathsf{rff}_j(f(\mathsf{dispatch}(s, i))).$$

Each universal quantifier can be replaced by a conjunction over all the values of $r$ and $j$, since the number of values is finite. This creates a number of separate formulas to check for each value of $r$ and $j$. However, these proofs will all be very similar to each other, and instead, we are going to reduce this plethora of proofs to just a few more general ones.

For each formula of the form $\forall x.\, \phi(x)$ we first apply *universal Skolemization* to eliminate the quantifier:

$$\frac{M \models \phi(a)}{M \models \forall x.\, \phi(x)},$$

where $a$ is a new *uninterpreted constant*, or a *Skolem constant*. This is a standard quantifier elimination rule used in theorem proving. It states that in order to prove $\forall x.\, \phi(x)$ in the model $M$, it is sufficient to prove $\phi(a)$ in $M$ for arbitrary interpretations of the constant $a$. Instead of checking every interpretation of $a$ (which would effectively be an expansion of the quantifier into the conjunction), in theorem proving this constant is left as a symbolic uninterpreted constant, and treated just as any other constant. The values of the terms that depend on $a$ may not always be computed, for example, $a = 0$, or $a + 1$, and such terms are also kept in their symbolic representation.

In general, in a successful proof of $\phi(a)$ a finite number of terms occur which depend on $a$ (by the mere reason that the proof is finite). Let us call these terms $t_1, \ldots, t_n$. For simplicity, we assume that these terms are all uninterpreted Skolem constants. If this is not true, and some term $t$ is more complex than just a constant, then we rewrite $\phi(a)$ in an equivalent form

$$\forall y.\, t = y \Longrightarrow \phi(a)[t/y]. \tag{1}$$

After applying Skolemization once again to eliminate $\forall y$, we obtain a new formula

$$t = b \Longrightarrow \phi(a)[t/b],$$

and the term $t$ is now replaced everywhere in the proof of $\phi(a)$ by the new Skolem constant $b$. This technique can be applied not only to the terms that depend on $a$, but also to other terms whose values are not known at the time of verification. These, in particular, include input variables and terms that depend on them.

As a heuristic, we assume that only equivalences of the uninterpreted terms matter in the proof of $\phi(a)$. That is, it may be important whether $a = b$ or not, but not the particular values of $a$ and $b$. Since we have to prove $\phi(a)$ for arbitrary interpretations of all Skolem constants, this means that we have to consider all the cases for the equalities. For instance, if we only have two Skolem constants $a$ and $b$, then we have to prove $\phi(a)$ when $a = b$, and when $a \neq b$. Since the actual values of $a$ and $b$ are irrelevant, we can assign them *abstract values*, e.g. $a = b = c_0$ in the first case, and $a = c_0$, $b = c_1$ in the second, where $c_0 \neq c_1$ are some constants. The model $M$ can now be rebuilt with $\{c_0, c_1\}$ as the new domain for $a$ and $b$ using *abstract interpretation*, which may result in significant reduction of the state space and make the model amenable to model checking. All together, these steps comprise a transformation that we call *Universal Case Splitting*. A variant of this transformation has been first introduced in the Mur$\phi$ model checker [16] and later used in the Cadence SMV implementation [20].

Although the model size can be reduced dramatically by the case splitting, the number of cases to verify may grow exponentially with the number of Skolem constants. Also, the state space may still be too big if the number of terms is large, due to the size of the abstract domain. One way to alleviate this problem is to consider only a few *important* terms, and assign the rest of the terms a special value $\perp$ which means an *arbitrary nondeterministic choice*. The same abstract interpretation mechanism can be used to construct the abstracted model, and the resulting abstraction is conservative for ACTL$^*$. The "important" terms in most cases will be chosen by the user. Some useful hints, however, can be easily generated automatically, and we will see how it is done in our example of Tomasulo's algorithm.

To illustrate the idea, let us consider the proof of Lemma 3.9, the inductive step in the proof of Lemma 3.8 stating that $f$ is indeed a completion function. The register file part of the lemma is:

$$\forall r.\ R_r(f(s)) = R_r(f(\mathsf{clock}(s))).$$

Expanding $f$ on the left (see figure 6) leads to

$$\begin{aligned}\forall r.\ &\textbf{if } \mathsf{tag}_r(s) \neq \emptyset \textbf{ then } \mathsf{op}\big(\mathsf{rs}_{\mathsf{tag}_r(s)}(s)\big) = R_r(f(\mathsf{clock}(s)))\\ &\textbf{else } R_r(s) = R_r(f(\mathsf{clock}(s)))\end{aligned}$$

This formula refers to the reservation station indexed by $\mathsf{tag}_r(s)$. This term depends on $r$, which will become a Skolem constant after Skolemization. Hence, the term will be a symbolic term in the proof, and we introduce a new quantified variable for it, as in

Formula (1):

$$\begin{aligned}&\forall r.\ \forall t.\ \mathsf{tag}_r(s) = t \Longrightarrow\\ &\qquad \textbf{if } t \neq \emptyset \textbf{ then } \mathsf{op}(\mathsf{rs}_t(s)) = R_r(f(\mathsf{clock}(s)))\\ &\qquad \textbf{else } R_r(s) = R_r(f(\mathsf{clock}(s)))\end{aligned}$$

After Skolemization, we only reference one register with its tag and one reservation station.[2] All the other parts of the state are irrelevant to the specification and can be automatically removed with the *cone of influence* reduction. This reduction determines the data dependencies among state variables and prunes out those that cannot influence the specification.

Although we have two Skolem constants in the final formula (for $r$ and $t$), the proof does not depend on whether $r = t$ or not (in our model, register numbers are never compared with reservation station numbers), and we only need to consider one abstract interpretation of $r$ and $t$. In other words, for this property, it is sufficient to prove it in a model with one register and a tag, and one reservation station. To finish the proof of the correctness of $f$, we have to prove a similar property for the reference file, which requires a few more terms to be split on. Therefore, the final configuration to model check the latter property on is slightly bigger and contains several reservation stations, but is still small enough to be manageable.

The commutative diagram is more complex, since it includes dispatching a new instruction:

$$\forall s.\ \forall i.\ I(s) \Longrightarrow \mathsf{exec}(f(s), i) = f(\mathsf{dispatch}(s, i)).$$

As before, we split the equality into comparison of individual registers and reference file entries:

$$\forall r.\ R_r(\mathsf{exec}(f(s), i)) = R_r(f(\mathsf{dispatch}(s, i))) \tag{2}$$

$$\forall p.\ \mathsf{rff}_p(\mathsf{exec}(f(s), i)) = \mathsf{rff}_p(f(\mathsf{dispatch}(s, i))), \tag{3}$$

and then expand $f$, exec and dispatch in the two equations and apply our case splitting technique, as we did in the proof of Lemma 3.9.

After expanding these functions in Eq. (2), we have one term $\mathsf{tag}_r(s)$ that depends on the quantified variable $r$ (which will become a Skolem constant), and three more terms of unspecified values: $\mathsf{scheduler}(s)$, $\mathsf{op}(i)$, and $\mathsf{op}(\mathsf{rs}_t(s))$. Again, we generate new quantified variables for them by rewriting (2) as

$$\begin{aligned}&\forall r.\ \forall t.\ \forall t_d.\ \forall p_t.\ \forall p_d.\\ &\qquad\qquad t = \mathsf{tag}_r(s) \quad\ \wedge\ t_d = \mathsf{scheduler}(s)\\ &\qquad\quad\ \wedge\ p_t = \mathsf{op}(\mathsf{rs}_t(s)) \wedge p_d = \mathsf{op}(i)\\ &\quad \Longrightarrow R_r(\mathsf{exec}(f(s), i)) = R_r(f(\mathsf{dispatch}(s, i))).\end{aligned}$$

When we eliminate the quantifiers, we obtain two Skolem constants indexing reservation stations ($t$ and $t_d$) and two constants for opcodes ($p_t$ and $p_d$). After expanding all the functions in the formula, we observe that the value of the register $r$ depends on whether

its tag $t$ points to the reservation station which is being used for dispatching ($t_d$). In the latter case, the reservation station $t$ does not change with dispatch, and the final value of the register $r$ is the opcode $p_t$ stored in that reservation station. If $t = t_d$, then the value of the register $r$ is the opcode $p_d$ of the instruction being dispatched. Either way, for each particular value of $r$, $t$, $t_d$, and $p_t$ the specification depends on at most one reservation station ($t$), one register ($r$) and two opcodes. All the other parts of the system can be removed by the cone of influence reduction, and we only have to model check this small part of the processor, independently of how many registers and reservation stations it contains. We only need to consider 4 different interpretations of the Skolem constants: 2 cases of whether $p_t = p_d$ or not, times 2 cases of whether $t = t_d$ or not.

The reference file part (Eq. (3)) is more complex and requires case splitting on more terms. First, we separate the comparison of the two arguments of the reference file entry into two separate formulas:

$$\forall p.\ \mathsf{arg}_1(\mathsf{rff}_p(\mathsf{exec}(f(s), i))) = \mathsf{arg}_1(\mathsf{rff}_p(f(\mathsf{dispatch}(s, i)))) \tag{4}$$

$$\forall p.\ \mathsf{arg}_2(\mathsf{rff}_p(\mathsf{exec}(f(s), i))) = \mathsf{arg}_2(\mathsf{rff}_p(f(\mathsf{dispatch}(s, i)))) \tag{5}$$

Each argument in a reference file entry is read from a reservation station holding its corresponding instruction. The instruction $i$ is either dispatched, or is already in the reservation station. If the arguments are not ready, their tags must be expanded, and one more reservation station will be referenced by each argument's tag. We only show the final formula for the first argument of a reference file entry (4), the other argument is handled similarly.

First, we split the cases on whether the entry's opcode is in the newly dispatched instruction, or already in some reservation station, or is not present at all (figure 11). In the last case, the reference file entry will not change, and it will receive some new value in the first two cases.

As before, after eliminating the quantifiers and expanding the functions in all three formulas, we end up using up to 2 reservation stations, one register and 2 opcodes. All the other parts of the machine become irrelevant for the specification and can be removed by the cone of influence reduction. Thus, we only need to model check three sets of specifications for this fixed size of the machine in order to prove the correctness of Tomasulo's algorithm in general.

## 6. Summary of the verification technique

Figure 12 gives a high-level view of the verification steps. Our primary goal is to show that the OOO machine is equivalent to the sequential (specification) machine for arbitrary instruction sequences. This is proven by induction over the instruction sequence. The base case, when the sequence is empty, is trivial. The induction step comprises the *Commutative Diagram* introduced by Burch and Dill [5]. To prove the commutative diagram, we use our new *Reference File encoding* of terms with uninterpreted function symbols. This encoding allows us to write a *Completion Function* $f$ which eliminates the need for expensive flushing operation. The commutative diagram property is then reformulated in terms of $f$. However,

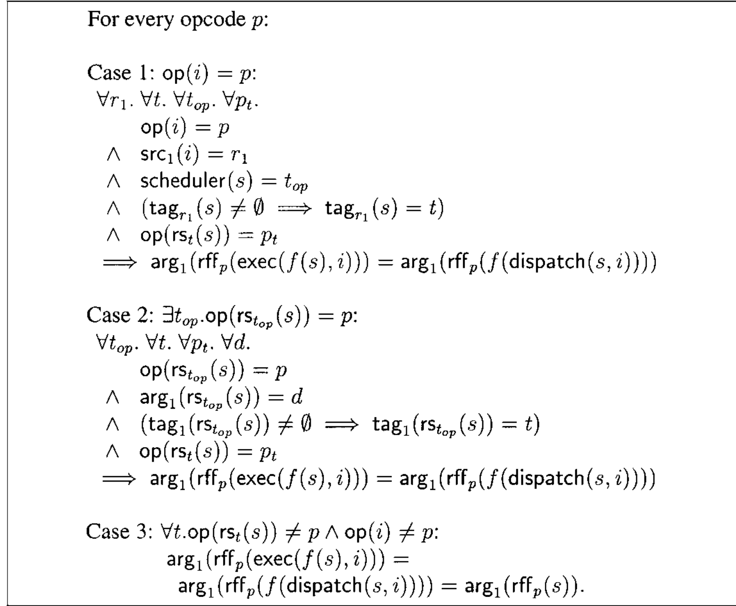


*Figure 11.* Splitting cases on the first argument of a reference file entry, Eq. (4).

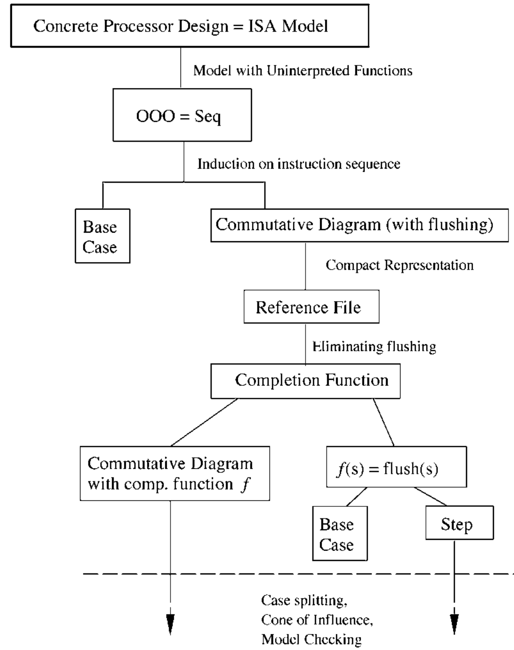


*Figure 12.* Structure of the verification procedure.

since $f$ is provided by the user, we also need to prove that $f$ is indeed a completion function, that is, it is equivalent to flushing. This is done by induction over the number of cycles required to flush the OOO machine. The base case is again simple, since we only need to check that $f(s) = s$ when $s$ is flushed. The induction step essentially shows that flushing the machine immediately using $f$ is the same as first running it for one clock cycle and then flushing it.

At the end of this stage, we are left with two properties: the commutative diagram in terms of $f$ in the reference file representation, and the induction step for the function $f$. These two properties are still hard to prove directly, since they heavily rely on the OOO model. We then apply *Case Splitting* transformations to them a number of times before the properties become small enough for direct application of model checking.

The splitting can also be automated by computing the *Cone of Influence* for the current specification and the model, and finding those state variables and terms that have uninterpreted values, or have infinite or too large finite types, and split on the values of these terms. Some human guidance might still be required to avoid unnecessary splits, but, in principle, the technique will work even without it.

## 7. Experimental results

We have implemented Tomasulo's algorithm using our reference file representation in the Cadence version of SMV. This version of SMV supports a limited amount of theorem proving together with symmetry and cone of influence reductions. After all the case splitting, as described in Section 5, SMV generates 50 sublemmas which all together verify in 120 seconds on a 450 MHz Pentium-II with 128 MB RAM. The proof of these 50 sublemmas accomplishes the partial correctness proof of Tomasulo's algorithm for an arbitrary configuration of the machine.

The SMV input specification involves the encoding of Tomasulo's algorithm, the definition of our light-weight completion function, Lemmas 3.7 and 3.9, and proof rules for case splitting as described in Section 5. No additional lemmas were needed. We believe that this new methodology can be extended to handle even more complex and realistic superscalar processor designs. We are currently building an automated proof assistant to facilitate this type of reasoning.

## 8. Conclusion and future directions

We have designed a methodology that provides generality and high degree of automation in verification of complex hardware devices, compared to the correspondingly general existing techniques (figure 13). It is based on our new method for constructing a completion function with a compact reference file representation. It also benefits from its combination with several existing techniques such as compositional model checking, theorem proving, and uninterpreted function symbols. We have demonstrated that our methodology makes it possible to verify complex parameterized designs almost completely automatically. The only manual effort required from the user is to write the completion function, which in our framework can be done relatively easily.

| feature \ approach | Ours | SMV | MC | TP |
|---|---|---|---|---|
| **design→model** | C | – | – | C- |
| **Induction** | C+ | C | – | C+ |
| **Case splitting** | A- | C | – | C |
| **MC (heavy case analysis)** | A | A | A | – |

*Figure 13*. Summary of supported features in different verification approaches, including ours. Legend: C = computer assisted under human guidance, C+ = interactive computer assistance, C− = computer assisted, but very tedious to do; A = automatic, A− = almost automatic; '–' = not supported. SMV = Cadence SMV (McMillan's approach), MC = Model Checking, TP = Theorem Proving.

As a benchmark, we take Tomasulo's algorithm for scheduling out-of-order instruction execution and demonstrate that it can be easily verified in our framework. The algorithm is parameterized by the processor configuration, and our approach allows us to prove its correctness in general, independent of the actual processor setup. For a typical specification we apply skolemization to eliminate universal quantifiers and perform straightforward symmetry reduction to obtain a propositional formula which is then verified by a model checker. Currently, the skolemization step is specified manually in the Cadence SMV model checker, and the rest of the verification is done automatically. However, the skolemization step can also be easily automated. We are currently developing a tool that will be able to perform this type of verification more automatically.

Another manual part of the verification process is developing and validating the abstract model. In our case, we represented the actual design with uninterpreted function symbols and nondeterministic oracles for the instruction scheduling mechanism. Although it is virtually impossible to automate this step completely, it can be greatly simplified by using a more sophisticated higher-level input language instead of SMV. The comparative table of features of such a tool with other existing tools and approaches is given in figure 13.

## Notes

1. In this paper, the instructions are restricted to the "arithmetic-style," taking 2 arguments and producing one result.
2. Of course, we also need to expand the other $f$ and clock, but this will not yield new register or reservation station in this case.

## References


1. S. Berezin, A. Biere, E. Clarke, and Y. Zhu, "Combining symbolic model checking with uninterpreted functions for out-of-order processor verification," in *FMCAD'98*, Lecture Notes in Computer Science, Vol. 1522, Springer-Verlag, Berlin, 1998, pp. 369–386.
2. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS'99*, Lecture Notes in Computer Science, Vol. 1579, Springer-Verlag, Amsterdam, The Netherlands, 1999.
3. R.E. Bryan, S. German, and M.N. Velev, "Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic," Technical Report, Carnegie Mellon University, 1999. Available as `http://reports-archive.adm.cs.cmu.edu/anon/1999/CMU-CS-99-115.ps`.
4. R.E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, Vol. 35, No. 8, pp. 677–691, 1986.

5. J.R. Burch and D.L. Dill, "Automatic verification of pipelined microprocessor control," in D.L. Dill (Ed.), *Computer Aided Verification (CAV'94)*, Lecture Notes in Computer Science, Vol. 18, Springer-Verlag, Berlin, 1994.
6. E. Clarke and E.A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proceedings of the IBM Workshop on Logics of Programs*, Springer-Verlag, Berlin, 1981, pp. 52–71.
7. E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 2, pp. 244–263, 1986.
8. *Computer Aided Verification (CAV'98)*, Lecture Notes in Computer Science, Vol. 1427, Springer-Verlag, Berlin, June 1998.
9. *Computer Aided Verification (CAV'99)*, Lecture Notes in Computer Science, Vol. 1633, Springer-Verlag, Berlin, July 1999.
10. W. Damm and A. Pnueli, "Verifying out-of-order executions," in D. Probst (Ed.), *CHARME'97*, Chapman & Hall, London, 1997.
11. L. Gwennap, "Intel's P6 uses decoupled superscalar design," *Microprocessor Report*, Vol. 9, No. 2, pp. 9–15, 1995.
12. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1996.
13. R. Hojati and R.K. Brayton, "Automatic datapath abstraction of hardware systems," in *Computer Aided Verification (CAV'95)*, Springer-Verlag, Berlin, 1995.
14. R. Hosabettu, M. Srivas, and G. Gopalakrishnan, "Decomposing the proof of correctness of pipelined microprocessors," in *Computer Aided Verification (CAV'98)*, Lecture Notes in Computer Science, Vol. 1427, Springer-Verlag, Berlin, June 1998, pp. 122–134.
15. R. Hosabettu, M. Srivas, and G. Gopalakrishnan, "Proof of correctness of a processor with reorder buffer using the completion function approach," in *Computer Aided Verification (CAV'99)*, Lecture Notes in Computer Science, Vol. 1633, Springer-Verlag, Berlin, July 1999.
16. C.N. Ip and D.L. Dill, "Better verification through symmetry," *Formal Methods in System Design*, Vol. 9, No. 1/2, pp. 41–75, 1996.
17. S.L.P. Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, Englewood Cliffs, NJ, 1987.
18. P.M. Kogge, *The Architecture of Symbolic Computers*, McGraw-Hill, New York, 1991.
19. K.L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic Publishers, Dordrecht, 1993.
20. K.L. McMillan, "Verification of an implementation of tomasulo's algorithm by compositional model checking," in *Computer Aided Verification (CAV'98)*, Lecture Notes in Computer Science, Vol. 1427, Springer-Verlag, Berlin, June 1998.
21. K. Sajid, A. Goel, H. Zhou, A. Aziz, and V. Singhal, "BDD based procedures for a theory of equality with uninterpreted functions," in *Computer Aided Verification (CAV'98)*, Lecture Notes in Computer Science, Vol. 1427, Springer-Verlag, Berlin, June 1998.
22. J. Sawada and W.A. Hunt, "Processor verification with precise exceptions and speculative execution," in *Computer Aided Verification (CAV'98)*, Lecture Notes in Computer Science, Vol. 1427, Springer-Verlag, Berlin, June 1998.
23. J.U. Skakkebæk, R.B. Jones, and D.L. Dill, "Formal verification of out-of-order execution using incremental flushing," in *Computer Aided Verification (CAV'98)*, Lecture Notes in Computer Science, Vol. 1427, Springer-Verlag, Berlin, June 1998.
24. M.N. Velev and R.E. Bryant, "Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions," in *Correct Hardware Design and Verification Methods (CHARME'99)*, Lecture Notes in Computer Science, Vol. 1703, Springer-Verlag, Berlin, 1999, pp. 37–53.
25. D.H.D. Warren, "An abstract prolog instruction set," Technical Note 309, SRI International, 1983.
26. P. Wolper, "Expressing interesting properties of programs in propositional temporal logic," in *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages (POPL'86)*, ACM, New York, 1986, pp. 184–193.