Verification of Data-Insensitive Circuits: An In-Order-Retirement Case Study*

A. Pnueli and T. Arons

Weizmann Institute of Science, Rehovot, Israel

Abstract. There is a large class of circuits (including pipeline and out-of-order execution components) which can be formally verified while completely ignoring the precise characteristics (e.g. word-size) of the data manipulated by the circuits. In the literature, this is often described as the use of *uninterpreted functions*, implying that the concrete operations applied to the data are abstracted into unknown and featureless functions. In this paper, we briefly introduce an abstract unifying model for such *data-insensitive* circuits, and claim that the development of such models, perhaps even a theory of *circuit schemas*, can significantly contribute to the development of efficient and comprehensive verification algorithms combining deductive as well as enumerative methods.

As a case study, we present in this paper an algorithm for out-of-order execution with in-order retirement and show it to be a refinement of the sequential instruction execution algorithm. Refinement is established by deductively proving (using PVS) that the register files of the out-of-order algorithm and the sequential algorithm agree at all times if the two systems are synchronized at instruction retirement time.

1 Introduction

As the complexity of hardware designs has grown, so has the need for advanced validation techniques. The need for formal verification tools to support industrial design processes is now recognized [22] and is apparent by the introduction of commercial verification tools. However, while circuit comparison and – to a lesser extent – property verification based on symbolic model-checking [5] have found their way into industrial applications (c.f. e.g. [10, 7]), coping with the complexity of industrial designs remains a key challenge, requiring complementary proof-methods to be combined in verification environments.

Obviously, no single verification method can tackle the entire design in one go, and it is necessary to reduce a large monolithic verification problem into an (often large) number of smaller problems. This reduction can be done in two complementary ways. A "bottom-up" approach partitions the design into sufficiently small units and attempts to verify each unit separately, while providing some approximate representation of the other units as a much simplified environment. This is the approach of *compositional verification* discussed in many

* This research was supported in part by a gift from Intel, a grant from the Minerva foundation, and an *Infrastructure* grant from the Israeli Ministry of Science.

articles such as [13], [1], [23], [27]. Traditionally, the separate verification of the individual units is performed by model checking, while using the locally established properties to infer global properties of the entire system is sometimes aided by deductive techniques.

A dual "top-down" approach handles a much larger part of the design (ideally the entire design) but reduces the complexity by abstracting away some of the detail, replacing some small circuits by an abstract operation, and bit-level data-structures by more abstract data types. Typically, we may need deductive techniques in order to justify and compute the abstractions, but often the abstracted system is small enough (and finite-state) so that it can be model-checked. Such use of abstraction is recommended and exemplified in [9], [20], [3], [12], [4], [11], and [15].

An important source of abstraction is that many components of a microprocessor design consist of an intricate controller managing the flow of data between various registers and arithmetical units, but their correctness is completely independent of the specific character of the flowing data. This gave rise to the idea that such circuits can be verified using uninterpreted (abstract) data domains and *uninterpreted functions* operating on them. When verifying such circuits, we usually *compare* the design against a *reference* machine and the only concern is that whenever the circuit applies an operation to (say) two arguments, then so does the reference machine, independently of the nature of the data or the operations. This idea has been successfully and effectively applied in [6], [14], [19], and the papers [8] and [31] in this volume.

The main principle in this particular abstraction is a clear separation between data and control, with the assumption that when the control part of the state is finite (and small or very regular) and the data remains uninterpreted, some verification and decision problems will become more tractable. A similar programme was undertaken in the early 70's in the study of programs in a sufficiently abstract way, leading to the theory of *program schemes* [26] which extensively studied an abstract model of (both imperative and declarative) programs assuming a simple (often finite) control, and uninterpreted data and operation. This theory proved very productive in recognizing some simple undecidable cases, as well as serving as an appropriate platform in which one can study program transformations, such as elimination of "go-to's" and mutual translations between recursion and iteration [18].

We believe that a similar study of abstract "circuit schemes" of particular classes can yield similar advantages and can serve as a useful platform for the development of various verification algorithms.

A first step in this direction has been taken in [17], by introducing the abstract model of a *Herbrand Automaton*. This model combines finite-state control with uninterpreted data and function registers, thus yielding a finite representation of infinite-state machines. Herbrand automata are used to model the proposed out-of-order (or pipeline) design as well as the reference machine which performs the same stream of instructions in a strictly sequential mode. The problem of verifying that a proposed out-of-order design correctly implements the reference model (which serves as a specification) is reduced to checking the equivalence of two Herbrand Automata.

To make this problem (which is undecidable in the general case) tractable, we restrict our attention to automata which apply each operation requested by an instruction only a bounded number of times. For such *Automata of Bounded Application* (HABA's), we show that the equivalence problem is decidable. In [17], we presented a decision algorithm for a general HABA, which is based on annotation of all reachable control states by invariants that hold on all visits to this location. For the special case of 1-bounded automata, we presented a more efficient decision algorithm, which is based on the comparison of the two automata over programs with operations taken from a limited set of four boolean operations.

It is interesting to note that the verification methods proposed by the papers [8] and [31] presented in this volume can, to some extent, be viewed as special cases of the general annotation method described in [17], where the additional abstractions explain their efficiency which is significantly higher than that of [17]. In retrospect, one can identify a narrower class of Herbrand Automata for which the more efficient approach can be justified in general. For example, one of the assumptions that simplifies the construction and management of annotations is that the reference machine is always "faster" than the out-of-order design, in the sense that every operation is performed by the reference machine *not later* than its execution by the OOO design.

In the rest of this paper, we present a case study of the verification of an out-of-order design with in-order retirement, which is verified using deductive techniques with the PVS tool.

2 Case Study: Out-of-Order Execution Designs

Modern out-of-order super-scalar microprocessors use dynamic scheduling to increase the number of instructions executed per cycle. These processors maintain a fixed-size window into the instruction stream, analyzing the instructions in the window to determine which can be executed out of order to improve performance. Branch prediction and register renaming are employed in order to keep the window full, while result-buffering techniques maintain the in-order-execution model required by the architecture.

In this paper we propose a proof-method for proving correctness of such processor designs based on *refinement*, and illustrate it by showing correctness of an algorithm for out-of-order execution with in-order-retirement. Our model is based on the Tomasulo algorithm in [34, 16] and [21], with modifications for in-order-retirement adapted from [25]. This paper is a continuation of the work on out-of-order execution presented in [16] and [2], extending the methodology to deal with in-order-retirement. As pointed out in [2] a correct choice of the stage in the instruction execution cycle at which to synchronize the out-of-order and sequential systems has a great impact on the complexity of the refinement relation and proof of correctness. Whereas synchronizing at instruction dispatch time proved efficient for Tomasulo's algorithm [2], in this paper we exploit the properties of in-order-retirement by synchronizing on instruction retirement. We share with [16] and [2] the ability to cope with *generic* designs, establishing correctness for an arbitrary number of functional units performing arbitrary arithmetic operations. Arithmetic operations are modeled as *uninterpreted functions*, allowing us to verify instruction scheduling without performing arithmetic verification.

Our methodology is intuitively simple: we force system IOR, exhibiting out-of-order execution with in-order-retirement, and the sequential system, SEQ, to retire instructions simultaneously. The in-order-retirement of system IOR implies that its register file is always compatible with the sequential execution of those instructions which have been retired. This fact is exploited by comparing only the systems' register files which, recording only retired instructions, are proved to be equal. In this way our refinement relation can largely ignore the complicated data structures in system IOR and any instructions which are in progress. We prove consistency of the two systems at every cycle during the execution, not only when execution has completed. Consistency between the data structures internal to system IOR is proved without reference to system SEQ. Verification is performed within the PVS [30] theorem prover.

Recent papers [28, 31–33] propose new techniques for hardware verification. In [28] Tomasulo's algorithm is verified using the SMV verifier, and an impressive level of automation is achieved. The proof, however, is dependent on the configuration and arithmetic operators and any modification requires that the new system be verified afresh.

In [31] symbolic model checking is used to verify Tomasulo's algorithm. A reference file containing symbolic values is used to implement uninterpreted functions, and abstraction techniques are developed to reduce the size of the model. The correctness of the method is proved using PVS. This approach is applicable only to designs of limited size, although this problem may be alleviated somewhat by combining it with other approaches.

A model of in-order-retirement is verified in [32] using the Stanford Validity Checker. A two-part approach is used: First it is shown that the implementation refines an intermediate in-order-execution abstraction. Functional equivalence between this abstraction and the specification system is then shown by incrementally flushing the pipeline and buffers in the intermediate abstraction.

In [33] a model including in-order-retirement, exceptions, and speculative instruction execution is verified. An intermediate model comprising a table of history variables is used to verify the system in ACL2. While the model verified is impressively detailed, the proof has the disadvantage of being specific to one configuration.

In comparison with the above works we believe our approach is both intuitive and straightforward, requiring neither specialized tools, intermediate models, nor flushing mechanisms. Our refinement relation is uncomplicated and its correctness is proved in a direct manner. The generality of the proof for all configurations and the use of uninterpreted functions are advantages shared with some of these papers.

We base this paper on the semantic model of *synchronous transition systems* ([24]), a variant of the clocked transition systems used in [29] in particular providing the concept of *steps*. Steps correspond to clock-cycles at the hardware-level. Transitions are expressed using first-order transition predicates.

The rest of the paper is structured as follows: The next section presents a short summary of the underlying mathematical model of synchronous transition systems and their refinement theory. Ultimately, the implementation has to be compatible with the sequential reference model presented in section 4. Section 5 introduces the out-of-order execution design. In section 6 we define the refinement relation, and in section 7 we prove the invariants needed by the refinement relation, thus completing the proof of correctness.

3 Synchronous Transition Systems and their Refinement

As our computational model we take synchronous transition systems ([24]). A *synchronous transition system* (STS) $S = \langle V, \Theta, \rho \rangle$, consists of the following components:

- V : A finite set of typed *system variables*. We define a *state* s to be a type-consistent interpretation of V, assigning to each variable $u \in V$ a value $s[u]$ over its domain. We denote by Σ the set of all states.
- Θ : The *initial condition*. A satisfiable assertion characterizing the initial states.
- ρ : A *transition relation*. This is an assertion $\rho(V, V')$, which relates a state $s \in \Sigma$ to its possible successors $s' \in \Sigma$ by referring to both unprimed and primed versions of the system variables. An unprimed version of a system variable refers to its value in s, while a primed version of the same variable refers to its value in s'.

Let $S = \langle V, \Theta, \rho \rangle$ be an STS. A *computation* of S is an infinite sequence of states σ: $s_0, s_1, s_2, \ldots$, satisfying the following requirements:

- *Initiation*: s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution*: State s_{j+1} is a S-successor of s_j, for each $j = 0, 1, \ldots$.

3.1 Refinement between Systems

Refinement is the comparison of an *abstract system* $S_A = \langle V_A, \Theta_A, \rho_A \rangle$ and a *concrete system* $S_C = \langle V_C, \Theta_C, \rho_C \rangle$. The abstract system serves as a *specification* capturing all the acceptable correct computations of the system. Correctness of the concrete system is established by proving that S_C *refines* S_A. Refinement means that every computation of S_C corresponds to some computation of S_A.

Let Σ_A and Σ_C denote the sets of abstract and concrete states respectively. Let Ω, referred to as the *domain of observations*, denote a set of elements. Let $\mathcal{O}_A : S_A \mapsto \Omega$ and $\mathcal{O}_C : S_C \mapsto \Omega$ be two functions termed the *abstract* and *concrete observation functions*, respectively. These functions indicate the parts of the systems which are compared in the refinement relations.

Given a system S and an observation function $\mathcal{O} : \Sigma \mapsto \Omega$, we define an *observation of* S to be an infinite sequence of Ω-elements obtained by applying $\mathcal{O}$ to a computation of S. Let $Obs_{\mathcal{O}}(S)$ denote the set of all observations of S according to the observation function $\mathcal{O}$.

Given systems S_A and S_C with observation functions $\mathcal{O}_C$ and $\mathcal{O}_A$ respectively, S_C *refines* S_A *according to the observation pair* $(\mathcal{O}_C, \mathcal{O}_A)$, denoted

$$S_C \sqsubseteq_{(\mathcal{O}_C, \mathcal{O}_A)} S_A,$$

if $Obs._{\mathcal{O}_C}(S_C) \subseteq Obs._{\mathcal{O}_A}(S_A)$.

When the identity of the observation functions is obvious form the context we simply write $S_C \sqsubseteq S_A$ to indicate that S_C refines S_A.

3.2 Verifying refinement

Given a concrete system $S_C = \langle V_C, \Theta_C, \rho_C \rangle$ with observation function $\mathcal{O}_C$, and an abstract system $S_A = \langle V_A, \Theta_A, \rho_A \rangle$ with observation function $\mathcal{O}_A$, such that $V_C \cap V_A = \emptyset$, we define an *interpolating system*

$$S_I = \langle V_C \cup V_A, \Theta_C \wedge \Theta_A, \rho_C \wedge \rho_A^* \rangle$$

where $\rho_A^*(V_C, V_C', V_A, V_A')$ may refer to all variables in $V_C \cup V_A$ in their primed and unprimed versions.

The intention of the interpolating system S_I is that it emulates the joint behavior of S_C and S_A in a way that would allow any previously admissible step of S_C and immediately match it with an S_A-step. Thus, $\rho_C \wedge \rho_A^*$ should not exclude any possible S_C-step, but is allowed to select among the possible S_A-steps a single one that matches the S_C-step. Intuitively, ρ_A^* is a modification of ρ_A taking as parameters V_C and V_C' in order to choose from the successors defined by ρ_A the one which matches the S_C-step. We further require that the projection of an S_I-computation onto V_A is a legal computation of S_A.

In any interpolating system S_I satisfying the above requirements the problem of showing that $S_C \sqsubseteq S_A$ is reduced to the problem of showing that $\mathcal{O}_C = \mathcal{O}_A$ is an invariant of S_I.

We formalize this as refinement rule REF:

$$\begin{array}{|l|}
\hline
\textbf{R1.}\ \exists V_A' : \rho_A^* \\
\textbf{R2.}\ \rho_A^* \longrightarrow \rho_A \\
\textbf{R3.}\ S_I \models \Box(\mathcal{O}_C = \mathcal{O}_A) \\
\hline
\qquad\qquad S_C \sqsubseteq S_A \\
\text{Rule REF: Proving refinement} \\
\hline
\end{array}$$

Premise R1 ensures that it is always possible to take a ρ_A^*-step, implying that the progress of system S_I is never blocked by ρ_A^*. Premise R2 requires that every

S_I-step is also an S_A-step. Premise R3 implies that the observation functions of S_C and S_A are equal for all states in all computations of S_I.

The three premises together ensure that, for every σ_C a computation of S_C, a computation of S_A can be found which completes σ_C into a computation σ_I of S_I. Since the equality of $\mathcal{O}_C$ and $\mathcal{O}_A$ is an invariant of the compound computation σ_I, the observation functions of S_A and S_C are equal at every point of σ_I, proving that S_C refines S_A.

In the proof of premise $R3$ one often uses auxiliary invariants of system S_I, most of which are also invariants of system S_C or of system S_A. To simplify the proofs of these invariants, we observe that every invariant of S_C or S_A is automatically an invariant of S_I.

4 The Reference Model: System SEQ

In this section we present system SEQ (Fig. 1) which is to serve as a reference model. System SEQ executes in a strictly sequential manner an input program consisting of non-branching register-to-register instructions. It accepts two parameters, N, the number of instructions, and R the maximum register index.

Instructions are stored in an array *prog* of length N. Each instruction has an *operation*, a *target* and two *source* operands. A program counter, *top*, points to the next instruction in *prog*. A register file *reg* records the current values of each register $0..R$.

At each step system SEQ either delays, in which case no change is made in the system, or executes the instruction pointed to by *top*. The value computed by the instruction is stored in $reg[prog[top].tgt]$ and *top* is incremented by one.

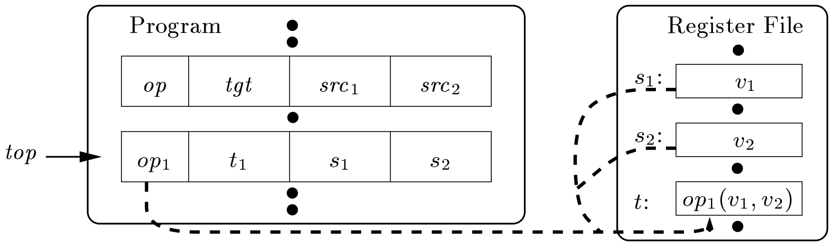

Fig. 1. Execution of one instruction in system SEQ

5 The Out-Of-Order Algorithm

In this section we provide a detailed description of our algorithm for out-of-order data-driven instruction execution with in-order-retirement. Our definitions are based on the descriptions in [21, 16] and [25].

Instructions flow from the instruction queue to the retirement buffer, where they assume their places in the queue for retirement, and the dispatch buffer, where they await availability of their source operands and a free execution unit. Once both operands are available execution of the instruction can be initiated by the appropriate functional unit. Results are written back to the retirement and dispatch buffers. Once an instruction reaches the head of the retirement queue it is retired, updating the register file with the instruction result.

In Fig. 2 we define system IOR, our implementation of the out-or-order execution, in-order-retirement algorithm. The data structures are illustrated in Fig. 3. We have added a number of auxiliary variables to our model in order to simplify the proofs. These variables are distinguished by underlining in Fig. 2. and by shading in Fig. 3. Auxiliary variables are only updated and copied from one record to another, so it is easy to prove that they do not affect the flow of control. The status field in the dispatch and retirement buffers is allowed to range over three values, **busy**, **write_b**, and **retire** instead of just busy and not-busy. However, the check in the algorithm is only on whether or not the value is busy, so this extended range does not influence the flow of control.

The functionality of system IOR can be divided into four subsystems:

- DISPATCH: This module dispatches instructions in program order.
- EXECUTE: This module executes instructions in the dispatch buffer.
- WRITE_BACK: This module writes back completed instructions.
- RETIRE: This module retires the buffer at the head of the retirement queue.

While only one instruction is dispatched or retired per cycle, modules EXECUTE and WRITE_BACK are parameterized by the number of functional units: when one of these modules in invoked, each functional unit in the system may execute or write-back results. Multiple instructions may be executed and written back in each cycle.

In practice these four subsystems operate concurrently. That is, in the same cycle all four can be invoked simultaneously. It can be verified in the PVS theorem prover that any concurrent execution of the four subsystems is equivalent to a four-step sequential execution of the subsystems in which each subsystem is executed once. We therefore consider each of the four systems separately, ignoring the possible interaction between them.

We have tried to be faithful to the design of the PentiumPro processor presented in [25]. However, we do not allow multiple instruction issue or retirement in a single cycle. We have also abstracted away the pipeline. However, the non-determinism in our model compensates for this – pipeline progress and delays are reflected by the non-deterministic rate of progress of instructions in our model. Our model admits all behaviors possible in a system in which the pipeline is explicitly modeled.

The data structures

In addition to the parameters N and R of system SEQ, system IOR expects the following parameters: U, the number of functional units; Z, the number of slots in the dispatch buffer; and B, the size of the retirement buffer.

system $\textsc{ior}(N, R, U, Z, B\colon \mathbb{N}^+)$ **is**
types
$REG\_ID = [0..R]$;
$FU\_ID = [1..U]$;
$PRODUCER = [0..B]$;
$VALUE =$ **floating-point-number** ;
$OP\_TYPE = \{\mathbf{fpadd}, \mathbf{fpsub}, \mathbf{fpmlt}, \mathbf{fpdiv}\}$;
$STATUS = \{\mathbf{busy}, \mathbf{write\_b}, \mathbf{retire}\}$;
$INST = [op\colon OP\_TYPE,\ tgt\colon REG\_ID,\ src\colon \mathbf{array}[1..2]\ \mathbf{of}\ REG\_ID]$;
$SRC\_TYPE = \mathbf{array}[1..2]\ \mathbf{of}$
$\quad [st\colon STATUS,\ \underline{reg\colon REG\_ID},\ prd\colon PRODUCER,\ v\colon VALUE\ ]$;
$DB\_TYPE = [\ oc\colon \mathbf{boolean},\ \underline{op\colon OP\_TYPE},\ \underline{tgt\colon REG\_ID},$
$\quad prd\colon PRODUCER,\ arg\colon SRC\_TYPE\ ]$;
$RB\_TYPE = [\ st\colon STATUS,\ \underline{op\colon OP\_TYPE},\ tgt\colon REG\_ID,$
$\quad v\colon VALUE,\ \underline{arg\colon SRC\_TYPE}\ ]$;
$RES\_TYPE = [active\colon \mathbf{boolean},\ \underline{tgt\colon REG\_ID},\ prd\colon PRODUCER, v\colon VALUE\ ]$;
variables
$prog\colon \mathbf{array}[0..N]\ \mathbf{of}\ INST$;
$RF\colon \mathbf{array}[0..R]\ \mathbf{of}\ VALUE\ \mathbf{init}\ 0$;
$RTT\colon \mathbf{array}[0..R]\ \mathbf{of}\ [busy : \mathbf{boolean},\ prd\colon PRODUCER\ ]\ \ \mathbf{init}\ \langle false, 0\rangle$;
$DB\colon \mathbf{array}[1..Z]\ \mathbf{of}\ DB\_TYPE\ \mathbf{init}$
$\quad \langle false,\ \mathbf{fpadd},\ 0,\ 0,\ \langle \mathbf{retire},\ 0,\ 0,\ 0\rangle, \langle \mathbf{retire},\ 0,\ 0,\ 0\rangle\rangle$;
$RB\colon \mathbf{array}[1..B]\ \mathbf{of}\ RB\_TYPE\ \mathbf{init}$
$\quad \langle \mathbf{retire},\ \mathbf{fpadd},\ 0,\ 0,\ \langle \mathbf{retire},\ 0,\ 0,\ 0\rangle, \langle \mathbf{retire},\ 0,\ 0,\ 0\rangle\rangle$;
$result\colon \mathbf{array}[1..U]\ \mathbf{of}\ RES\_TYPE\ \mathbf{init}\ \langle false,\ 0,\ 0,\ 0\rangle$;
$top\colon [1..N{+}1]\ \mathbf{init}\ 1$;
$next, oldest\colon PRODUCER\ \mathbf{init}\ 1$;
functions
$fu\_table\colon OP\_TYPE \mapsto FU\_ID$;
$succ\colon PRODUCER \mapsto PRODUCER$;
$occ\colon PRODUCER \mapsto \mathbf{boolean}$;
behavior

$$\textsc{dispatch}\ \ \Big\|\ \ \mathop{\Big\|}_{FU=1}^{U}\ \textsc{write\_back}[FU]\ \ \Big\|\ \ \mathop{\Big\|}_{FU=1}^{U}\ \textsc{execute}[FU]\ \ \Big\|\ \ \textsc{retire}$$

end system

Fig. 2. System IOR

The register file (denoted by RF) and register translation table, RTT, are both simple arrays indexed by the register index. The register file contains the current value for the register. The register translation table records whether any instruction targeting the register is in progress, and if so, which retirement buffer refers to the most recent such instruction.

The result array, *result*, stores the values of computations until these are written back.

The dispatch buffer, DB, is an unordered array of slots, each containing a flag indicating whether it is occupied. This buffer contains all the information needed to perform the operation, and is accessed by the functional units when executing instructions.

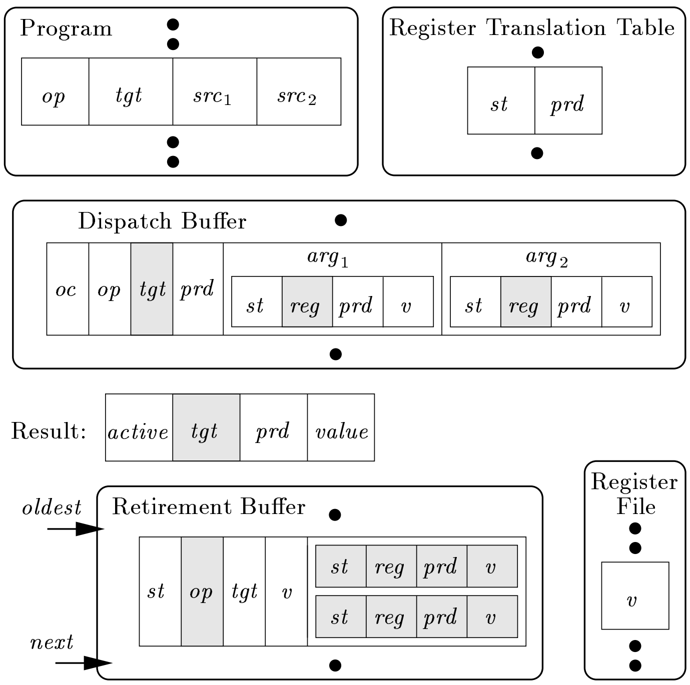

Fig. 3. Data structures for IOR

The retirement buffer, *RB*, is a circular array in which instructions are ordered according to their dispatch order. Its function is to ensure that instructions are retired in-order. The buffer is treated as a queue, with the the oldest buffer being "popped" off during retirement, while dispatched instructions are "pushed" onto the end of the queue. The pointers *oldest* and *next* point to the head of the queue and the next free buffer, respectively. The retirement buffer is empty when *next* and *oldest* point to the same slot. We never allow all the slots in the buffer to be occupied. The function *succ* defines a buffer's successor in the circular array, while *occ* determines (using *next* and *oldest*) whether a buffer is occupied.

The four subsystems

System DISPATCH: Instructions are *issued* from the instruction stream to buffers in both the dispatch and retirement buffers. In the dispatch buffer any unoccupied slot S_n may be used. In the retirement buffer the next slot in the queue, that pointed to by *next*, is used. Since we do not allow all the slots in the retirement buffer to be occupied, we dispatch only when the successor of *next* is not *oldest*. Operand registers are looked up in the register translation table which

system DISPATCH **is**
definitions
$\langle o, t, s_1, s_2 \rangle = prog[top]$
$S_n = \textbf{choose } S{:}\,[1..Z];$
$delay = \textbf{choose } d \in \{true, false\};$
$\forall\, j : \{1, 2\}$
$p_j = RTT[s_j].prd;$
$arg_j = \textbf{if } RTT[arg_j].busy \textbf{ then } \langle\, RB[p_j].st,\ s_j,\ p_j,\ RB[p_j].v \,\rangle;$
$\textbf{else } \langle\, \textbf{retire},\ s_j,\ 0,\ RF[s_j] \,\rangle;$
behavior
if $top \leq N \ \wedge\ \neg DB[S_n].oc \ \wedge\ succ(next) \neq oldest \ \wedge\ \neg delay$ **then**
$DB[S_n] := \langle\, true,\ o,\ t,\ next,\ arg_1,\ arg_2 \,\rangle;$
$RB[next] := \langle\, \textbf{busy},\ o,\ t,\ 0,\ arg_1,\ arg_2 \,\rangle;$
$top := top + 1;$
$next := succ(next);$
end if
end system

records whether a value for the register is available in the register file (the status of the RTT entry is "not-busy") or, when the entry is busy, which buffer in the RB will produce the value for the register. If an operand's status is "not-busy", its value is simply copied from the register file to the dispatch buffer. Otherwise, data from the buffer producing the operand are copied from the operand's buffer in the RB to the operand field of the dispatch buffer. The operand also stores the index of the retirement buffer thus effecting register renaming – this field, rather than the index of the source register, is used to obtain the correct value for the operand. It uniquely defines the correct value for this operand, while the source register may be targeted by a number of other preceding and succeeding instructions. In our implementation operand information is also copied to auxiliary variables in the retirement buffer.

System EXECUTE**:** The dispatch buffer continuously snoops the bus, waiting for the values of any pending operands to be written back. Once both operands of an instruction are available the instruction may be *executed*. In our model the instruction operation determines which functional unit will eventually execute the instruction. The table *fu_table* maps operations to functional units. On each cycle each functional unit can execute an instruction if its result record is inactive. (An active result record indicates that a previously executed instruction is waiting to be written back.) On execution the result and retirement buffer index are passed to the *result* record of the functional unit and the dispatch buffer entry is freed. The active bit of the result record is set indicating that it is occupied and is waiting for access to the bus.

system EXECUTE(FU: [1..U]) **is**
definitions
$enabled(S$: [1..Z]): **boolean**
$= DB[S].oc \ \wedge \ DB[S].arg[1].st \ \neq$ **busy** $\wedge \ DB[S].arg[2].st \ \neq$ **busy**;
$e \quad =$ **choose** S: [1..Z];
$delay =$ **choose** $d \in \{true, false\}$;
behavior
if $enabled(e) \ \wedge \ fu\_table[DB[e].op] = FU \ \wedge \ \neg result[FU].active \ \wedge \ \neg delay$ **then**
$result[FU] := \langle true, DB[e].tgt, DB[e].prd,$
$do\_op\,(\,DB[e].op,\ DB[e].arg[1].v,\ DB[e].arg[2].v\,)\rangle;$
$DB[e].oc \quad := false;$
end if
end system

system WRITE_BACK(FU: [1..U]) **is**
definitions
$\langle\ active_{FU},\ tgt_{FU},\ prd_{FU},\ val_{FU}\ \rangle = result[FU]$
$delay$ = **choose** $d \in \{true, false\}$;
behavior
if $result[FU].active \ \wedge \ \neg delay$ **then**
$\langle\ RB[prd_{FU}].st,\ RB[prd_{FU}].v\ \rangle := \langle$ **write_b**, $val_{FU}\ \rangle$
$\forall\ b : [1..B], S : [1..Z], j : \{1..2\},$
if $occ(b) \ \wedge \ RB[b].arg[j].st =$ **busy** $\wedge \ RB[b].arg[j].prd = prd_{FU}$
then $\langle\ RB[b].arg[j].st,\ RB[b].arg[j].v\ \rangle \ := \langle$ **write_b**, $val_{FU}\ \rangle$;
if $DB[S].oc \ \wedge \ DB[S].arg[j].st =$ **busy** $\wedge \ DB[S].arg[j].prd = prd_{FU}$
then $\langle\ DB[S].arg[j].st,\ DB[S].arg[j].v\ \rangle := \langle$ **write_b**, $val_{FU}\ \rangle$;
end if
end system

System WRITE_BACK**:** Results in the result record are *written back* by placing the result and its tag (producer field) on the bus. The retirement buffer matching the tag on the bus records the new value for the instruction and changes its status to **write_b**. Both the dispatch and retirement buffers use values on the bus to update their operand fields.

System RETIRE**:** Once it reaches the top of the retirement queue (it is pointed to by *oldest*) a buffer which has a valid result (its status is not **busy**) can be *retired*. During retirement the value stored in the buffer is copied to the register file. If the register translation table points to the retiring buffer the busy-flag of the relevant entry is reset, indicating that the most current value can be found in the register file. In our implementation the status of operands in the dispatch and retirement buffers is also updated to reflect retirement.

```
system RETIRE is
definitions
 delay = choose d ∈ {true, false};
behavior
  if occ(oldest) ∧ RB[oldest].st ≠ busy ∧ ¬delay then
    RF[RB[oldest].tgt] := RB[oldest].v;
    if RTT[RB[oldest].tgt].prd = oldest then RTT[RB[oldest].tgt].busy := false;
    ∀ b : [1..B], S : [1..Z], j : {1..2},
      if   occ(b) ∧ RB[b].arg[j].st ≠ busy ∧ RB[b].arg[j].prd = oldest
      then RB[b].arg[j].st := retire;
      if   DB[S].oc ∧ DB[S].arg[j].st ≠ busy ∧ DB[S].arg[j].prd = oldest
      then DB[S].arg[j].st := retire;
  end if
end system
```

6 System IOR refines system SEQ

In this section we commence the proof that system IOR refines system SEQ. Our proof is dependent on invariants which we will prove in section 7.

Both system SEQ and system IOR contain a program counter *top*, so we denote its SEQ-instance by top_A and its instance in IOR by top_C. We ensure that the two systems retire instructions simultaneously. System IOR may store in its retirement buffer instructions which have been issued but not yet retired and thus top_C will exceed top_A by the number of occupied slots in the retirement buffer. We define the function $adj\_top(top_C,\ next,\ oldest)$, which adjusts top_C by the number of outstanding instructions, returning the instruction index of the next buffer to be retired. We use adj_top and adj_top' as abbreviations for $adj\_top(top_C, next, oldest)$ and $adj\_top(top_C{}', next', oldest')$, respectively.

The synchronization of the two systems at retirement time implies that adj_top always equals top_A. Correctness is shown by proving that the same value has been retired into the same register in the register files RF and reg of IOR and SEQ, respectively.

Our observation functions are:

$$\begin{array}{l} \mathcal{O}_A : (reg, top_A) \\ \mathcal{O}_C : (RF, adj\_top) \end{array}$$

The initial conditions for the systems are:

$$\begin{array}{l} \Theta_A : top_A = 1 \ \wedge \ \forall r : [0..R], reg[r] = 0 \\ \Theta_C : \left(\begin{array}{l} \wedge \ top_C = 1 \ \wedge \ next = oldest \\ \wedge \ \forall r : [0..R],\ RF[r] = 0 \ \wedge \ RTT[r].busy = false \\ \wedge \ \forall S : [1..Z],\ DB[FU,S].oc = false \\ \wedge \ \forall FU : [1..U],\ result[FU].active = false \end{array} \right) \end{array}$$

That is, we assume that both computations start at the first instruction with all registers storing the value zero, all dispatch and retirement buffers unoccupied.

For ρ^*_A, the restricted version of $\rho_{_A}$, we take

$$
\begin{array}{l}
\textbf{if} \quad top_{_C} \leq N \ \wedge \ adj\_top' \neq adj\_top \\
\textbf{then } top_{_A}{}' = top_{_A} + 1 \ \wedge \\
\quad reg' = \lambda r \in [0..R] \ \textbf{if} \quad r = prog[top_{_A}].tgt \\
\qquad\qquad\qquad\qquad\quad \textbf{then } do\_op\,(prog[top_{_A}].op, \\
\qquad\qquad\qquad\qquad\qquad\qquad reg[prog[top_{_A}].src[1]], reg[prog[top_{_A}].src[2]]) \\
\qquad\qquad\qquad\qquad\quad \textbf{else } \ reg[r] \\
\textbf{else } \ top_{_A}{}' = top_{_A} \ \wedge \ reg' = reg
\end{array}
$$

Thus ρ^*_A, the restricted transition relation for $S_{_A}$, simply states that SEQ executes one instruction whenever an instruction is retired by system IOR.

To prove refinement we now prove that the three premises $R1, R2$ and $R3$ of rule REF hold in our system.

Premise $R1$ holds trivially by definition of ρ^*_A.

To prove $R2$ we note that the definition of ρ^*_A is identical to that of $\rho_{_A}$ except that the non-deterministic choice in $\rho_{_A}$ of whether to delay is decided deterministically.

Premise $R3$ asserts that $\alpha : \mathcal{O}_{_C} = \mathcal{O}_{_A}$ is an invariant of S_I. In order to prove the invariance of α we need to show that α holds at the initial state in S_I and that whenever α holds at the current S_I-state it will hold at the next S_I-state.

Showing that α holds at the initial state is trivial.

Assume that α holds in the current state of S_I. Thus, $adj\_top = top_{_A}$ and the values in RF match the values in reg.

If system IOR does not retire an instruction in this step then SEQ delays. None of the registers in system IOR are changed and neither are the values in reg. Although $top_{_C}$ changes when an instruction is issued, adj_top does not. Thus, α holds at the next state in S_I.

When an instruction is retired by IOR adj_top is incremented by one. System SEQ will complete an instruction, incrementing its program counter, so $adj\_top' = top_{_A}{}'$. We assume that before retirement occurred the register files of the two systems agreed. Thus we know that the two register files recorded the same values for the operands, and it suffices to prove that the correct value is written to register $prog[adj\_top].tgt$ without affecting other registers:

$$
\begin{array}{ll}
\varphi_1 : & \left(\begin{array}{rcl} RF'[prog[adj\_top].tgt] & = & do\_op\,(\ prog[adj\_top].op, \\ & & \qquad\qquad RF[prog[adj\_top].src[1]], \\ & & \qquad\qquad RF[prog[adj\_top].src[2]]\ ) \end{array} \right) \\
& \wedge \qquad (\forall r : [0..R], r \neq prog[adj\_top].tgt \longrightarrow RF'[r] = RF[r])
\end{array}
$$

This property of system IOR is proved in section 7.

7 Some invariants of system IOR

In this section we outline the proofs of some of the invariants needed to prove the refinement relation.

We ultimately want to prove property φ_1 which asserts that the correct values are retired to the correct registers. Prior to its retirement the value in $RF'[prog[adj\_top].tgt]$ was stored in the retirement buffer pointed to by *oldest*. It is easy to prove that only register $RB[oldest].tgt$ is modified during retirement, so it suffices to prove that

$$\varphi_2 : \begin{pmatrix} RB[oldest].v \ = do\_op\,(\,prog[adj\_top].op,\, RF[prog[adj\_top].src[1]], \\ \qquad\qquad\qquad\qquad\qquad\qquad RF[prog[adj\_top].src[2]]\,) \end{pmatrix} \\ \wedge \qquad\qquad\qquad\qquad\qquad RB[oldest].tgt = prog[adj\_top].tgt$$

It is not difficult to prove that $RB[oldest].tgt = prog[adj\_top].tgt$. We thus concentrate on proving that the value retired is correct.

We store as auxiliary variables the instruction operation and the source registers of the two operands. The invariants

$$RB[oldest].op = prog[adj\_top].op$$
$$\text{and} \;\; RB[oldest].arg[j].reg = prog[adj\_top].src[j]$$

are easy to prove.

So, to prove φ_2 it suffices to prove that for any occupied buffer rb

$$\varphi_3 : \begin{pmatrix} RB[rb] = \langle\ \neg\mathbf{busy}, tgt, v, op, \langle st_1, prd_1, reg_1, v_1, \rangle, \\ \qquad\qquad\qquad\qquad\qquad \langle st_2, prd_2, reg_2, v_2, \rangle\ \rangle \end{pmatrix} \longrightarrow \\ st_1 \neq \mathbf{busy} \wedge st_2 \neq \mathbf{busy} \wedge v = do\_op(op, v_1, v_2)$$

providing that we also prove that the values stored in v_1 and v_2 are correct, i.e. these values will be found in registers reg_1 and reg_2 when buffer rb is retired.

We will now prove the correctness of the operands and then prove φ_3.

Tuples of the form $\langle st, prd, v\rangle$ occur repeatedly in the system and we call them records of type *TRIPLE_TYPE*. These records allow us to trace the progress of an instruction and match it to any operand fields which are waiting the completion of the instruction.

A *TRIPLE_TYPE* record is said to be *visible* if it is part of the main field of an occupied retirement buffer or one of its operands, or is part of one of the operand fields of an occupied dispatch buffer.

It can be shown that

$$\varphi_4 : visible(\langle st_1, prd_1, v_1\rangle) \wedge visible(\langle st_1, prd_1, v_2\rangle) \longrightarrow \;\; v_1 = v_2$$

That is, if two visible *TRIPLE_TYPE* records point to the same buffer and have the same status, they have the same value. The check on the status is necessary as when an entry in the retirement buffer is retired and a new instruction is placed in the vacated slot, operand fields in the system may use the same producer tag to refer to both the old and the new instructions.

We can prove that for any occupied buffer rb

$$\varphi_5 : \begin{pmatrix} RB[rb] = \langle\ \neg\mathbf{busy}, tgt, v, op, \langle st_1, prd_1, reg_1, v_1, \rangle, \\ \qquad\qquad\qquad\qquad\qquad\qquad \langle st_2, prd_2, reg_2, v_2, \rangle\ \rangle \end{pmatrix} \longrightarrow$$
$$\begin{array}{ll} \mathbf{if} & st_j = \mathbf{retire} \\ \mathbf{then} & \forall rb' : rb' \prec rb \longrightarrow RB[rb'].tgt \neq reg_j \\ \mathbf{else} & prd_j \prec rb \ \wedge\ RB[prd_j].tgt = reg_j \ \wedge \\ & \quad \forall rb' : prd_j \prec rb' \prec rb \longrightarrow RB[rb'].tgt \neq reg_j \end{array}$$

where $rb_1 \prec rb_2$ is true when both rb_1 and rb_2 are occupied and buffer rb_1 precedes buffer rb_2 in the queue defined by the *next* and *oldest* pointers.

Invariant φ_5 states that an operand field whose status is not "retired" points to the closest preceding retirement buffer with the correct target. If the operand is retired then there is no buffer preceding rb in the retirement buffer with the same target as the operand. This property allows us to assert that the value in the buffer prd_j will be retired to $RF[reg_j]$ before buffer rb is retired, and will not be overwritten before the retirement of rb.

Combining φ_4 and φ_5 it can easily be seen that for any occupied buffer rb

$$\varphi_6 : \begin{pmatrix} RB[rb] = \langle\ \neg\mathbf{busy}, tgt, v, op, \langle st_1, prd_1, reg_1, v_1, \rangle, \\ \qquad\qquad\qquad\qquad\qquad\qquad \langle st_2, prd_2, reg_2, v_2, \rangle\ \rangle \end{pmatrix} \longrightarrow$$
$$\begin{array}{llll} \mathbf{if} & st_j = \mathbf{retire} & \mathbf{then} & v_j = RF[reg_j].v \\ \mathbf{else\ if} & st_j = \mathbf{write\_b} & \mathbf{then} & v_j = RB[prd_j].v \end{array}$$

This, together with the precedence relation defined in φ_5, is the operand correctness property needed in φ_3. A simple corollary of φ_5 and φ_6 is that the values in the operand fields of the buffer pointed to by *oldest* are the values found in the register file.

We can now prove φ_3. Proving that the operand fields are not busy is easy. We need to show that the value in a buffer which is not busy can be obtained by applying the buffer operation to the operand values. The value v in $RB[rb]$ was obtained when an active result with target prd was written back. This result itself was calculated in a reservation station before being copied to the result register. By definition of our transition relation,

$$\begin{pmatrix} \wedge\ result'[FU] = \langle true, t, p, v\rangle \\ \wedge\ DB[S] = \langle true, t, p, op,\ \langle st_1, prd_1, reg_1, v_1\rangle, \langle st_2, prd_2, reg_2, v_2\rangle\ \rangle \end{pmatrix} \rightarrow$$
$$v = do\_op(op, v_1, v_2)$$

That is, the value calculated by the dispatch buffer is copied to the result register together with the producer and target fields. Proving that the operation in $DB[S]$ is the same as that in $RB[tgt]$ is not difficult. Using φ_4 it can be shown that $\forall\, j : \{1, 2\}, v_j = RB[tgt].args[j].v$. Thus, the value copied to $RB[tgt]$ equals the result of applying its operation to the values of its operands.

This completes the proof of φ_3. Invariants φ_3 and φ_6 allow φ_2 and φ_1 to be proved, completing the proof of the refinement relation.

8 Conclusion

We have presented a refinement-based proof-method for the verification of modern processor architectures, and demonstrated its applicability by showing the correctness of a data-path involving multiple functional units, register-renaming, dynamic scheduling, and out-of-order execution with in-order-retirement.

Our decision to synchronize the two systems at instruction retirement has obviated the need for an intermediate model or flushing, simplifying the refinement relation and its proof. The strength of our proof is its generality: we prove correctness for arbitrary configurations of unlimited size. The proof is also independent of the operations appearing in the instructions, and checks only the correctness of the out-of-order scheduling, not the calculation of mathematical expressions.

Still, this work needs to be extended in a number of ways:

- We would like to increase the degree of automation in the proofs. We are currently investigating the possibility of making better use of PVS strategies. Compilation of the employed notation for STS could ideally replace the current hand-translation.
- Our model is limited to non-branching programs in which no loads, stores or exceptions occur. Ongoing work considers extending the framework of this paper to incorporate some of these features.

References

1. M. Abadi and L. Lamport. Composing specifications. *Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness*, LNCS-430:1–41, 1990.
2. T. Arons and A. Pnueli. Verifying tomasulo's algorithm by refinement. Technical report, Weizmann Institute, 1998.
3. S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Properties preserving simulations. *CAV'92*:251–263, 1992.
4. N. Bjørner, I.A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. 1^{st} *Intl. Conf. on Principles and Practice of Constraint Programming*, LNCS-976:589–623, 1995.
5. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. and Comp.*, 98(2):142–170, 1992.
6. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. *CAV'94*:68–80, 1994.
7. G. Barrett and A. McIsaac. Model-checking in a microprocessor design project. *CAV'97*, 1997.
8. R.E. Bryant and M. Velev. Deciding a theory of positive equality with uninterpreted functions. This volume.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL'77*.
10. Y.A. Chen, E.M. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J.OLeary, and X. Zhao. Verification of all circuits in a floating point unit using word-level modelchecking. *FMCAD'96*:1–18, 1996.

11. E.M. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. *CONCUR'95*:395–407, 1995.
12. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Trans. Prog. Lang. Sys.*, 16(5):1512–1542, 1994.
13. E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. *Proc. 4th IEEE Symp. Logic in Comp. Sci.*:353–362, 1989.
14. D. Cyrluk and P. Narendran. Ground temporal logic: A logic for hardware verification. *CAV'94*:247–259, 1994.
15. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Prog. Lang. Sys.*, 19(2), 1997.
16. W. Damm and A. Pnueli. Verifying out-of-order executions. *CHARME'97*:23–47, Montreal, 1997. Chapmann & Hall.
17. W. Damm, A. Pnueli, and S. Ruah. Herbrand automata for hardware verification. *CONCUR'98*, 1998.
18. S. Greibach. *Theory of program structures: schemes, semantics, verification*, volume 36 of *Lect. Notes in Comp. Sci.* Springer-Verlag, Heidelberg, 1975.
19. R. Hojati, A. Isles, D. Kirkpatrick, and R.K. Brayton. Verification using uninterpreted functions and finite instantiations. *FMCAD'96*:218 – 232, 1996.
20. N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7):523–543, 1992.
21. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1996.
22. K. Keutzer. The need for formal methods for integrated circuit design. *FMCAD'96*:1–18, 1996.
23. R.P. Kurshan and K.L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.
24. Y. Kesten and A. Pnueli. An αSTS-based common semantics for SIGNAL, STATECHART, DC+, and C. Tech. report, Weizmann Institute, 1996.
25. Gwennap L. Intel's p6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, 1995.
26. D.C. Luckham, D.M.R. Park, and M.S. Paterson. On formalized computer programs. *J. Comp. Sys. Sci.*, 4(3):220–249, 1970.
27. K.L. McMillan. A compositional rule for hardware design refinement. *CAV'97*.
28. K.L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. *CAV'98*:110–121, 1998.
29. Z. Manna and A. Pnueli. Clocked transition systems. *Logic and Software Engineering*:3 – 42. World Scientific, Singapore, 1996.
30. S. Owre, J.M. Rushby, N. Shankar, and M.K. Srivas. A tutorial on using PVS for hardware verification. *Proceedings of the Second Conference on Theorem Provers in Circuit Design*:167–188. FZI Publication, Universität Karlsruhe, 1994.
31. E. Clarke S. Berezin, A. Biere and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-or-order processor verification. This volume.
32. J.U. Skakkebaek, R.B. Jones, and D.L. Dill. Formal verification of out-of-order execution using incremental flushing. *CAV'98*:pp 98–110, 1998.
33. J. Sawada and Jr. W.A. Hunt. Processor verification with precise exceptions and speculative execution flushing. *CAV'98*:135–146, 1998.
34. R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. of Research and Development*, 11(1):25–33, 1967.