Int J Softw Tools Technol Transfer (2003) 4: 298–312 / Digital Object Identifier (DOI) 10.1007/s10009-002-0087-0

A framework for superscalar microprocessor correctness statements

Mark D. Aagaard[1], **Byron Cook**[2], **Nancy A. Day**[3], **Robert B. Jones**[4]

[1] Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada
E-mail: m.aagaard@ece.uwaterloo.ca
[2] Prover Technology, Portland, Ore., USA; E-mail: byron@prover.com
[3] Computer Science, University of Waterloo, Waterloo, Ontario, Canada; E-mail: nday@cs.uwaterloo.ca
[4] Strategic CAD Labs, Intel Corporation, Hillsboro, Ore., USA; E-mail: rjones@ichips.intel.com

Published online: 17 December 2002 –

Abstract. Most verifications of superscalar, out-of-order microprocessors compare state-machine-based implementations and specifications, where the specification is based on the instruction-set architecture. The different efforts use a variety of correctness statements, implementations, and verification approaches. We present a framework for classifying correctness statements about safety properties of superscalar microprocessors. Our framework is independent of the implementation representation and verification approach, and is parameterized by the width of the processor. We characterize the relationships between the correctness statements of many different efforts and also illustrate how classical approaches to microprocessor verification fit within our framework.

Keywords: Microprocessor correctness – Commuting diagrams – Formal verification – Pipelines

1 Introduction

The increased parallelism provided by *out-of-order execution* in microprocessors has made correctness statements for verification complicated, varied, and even controversial. We studied published verifications of out-of-order microprocessors and discovered a wide variety of correctness statements, verification techniques, and processor implementations. Some correctness statements initially appear to be similar, such as the ones based on Burch-Dill style flushing [13], but differences emerge after close examination. Other statements are difficult to compare at first, but later reveal similarities. The goal of this paper is to provide a foundation for clarifying the meaning of individual correctness statements; for precisely comparing different statements; and for analyzing the interaction between processor features, verification strategy, and correctness statements.

Most recent verification efforts verify state-machine-based representations of microarchitectural implementations and instruction-set architectures. The verification efforts focus on safety – any behavior of the implementation is also a behavior of the specification. Liveness is usually dealt with as a secondary concern. In keeping with these trends, we focus on safety-based correctness statements that relate microarchitectural implementations to instruction-set architectures. We include deterministic and non-deterministic state machines with finite and infinite state spaces. We do not include specifications that are collections of properties, such as the work by Beatty and Bryant [8], McMillan [25], and Patankar et al. [30].

The result of our investigation and analysis is a framework that precisely describes and classifies correctness statements about safety between state machines. It allows correctness statements to be analyzed independent of verification techniques and microarchitectural features. In this paper, we introduce the framework, present its mathematical basis, and describe how published microprocessor correctness statements fit within the framework. This presentation is a revised and extended treatment of our original work [1]. In particular, we include superscalar microprocessors by parameterizing the correctness statements by the width of the processor. Additionally, we have precisely defined the relationships between the different correctness statements.

2 Modeling with state machines

We assume that both the specification and implementation have program memories as part of their state. Therefore, our state machines do not take instructions as inputs. Approaches that take instructions as inputs in their correctness statements (e.g. [9, 13, 22]) can be augmented with program memories that produce the input trace. In-

terrupts can also be treated as part of the state space by adding appropriate control circuitry to read the interrupt input trace from an internal store. We assume that state machines generate infinite traces, where "termination" of a program is denoted by repeating the final state of the program. Definition 1 shows the formalism we use to describe state machines.

Definition 1 (State machines)
A state machine M is a triple (Q, Q°, N) where:

- Q is the set of possible state values and is a Cartesian product of internal (hidden) state components and externally-visible state components.
- $Q^\circ \subseteq Q$ is the set of initial states.
- $N \subseteq Q \times Q$ is the next-state relation.

In addition, we use the following notation:

Q_e is the set of possible externally-visible state values.
$\Pi_\mathrm{e} : Q \to Q_\mathrm{e}$ is the corresponding projection function.
$q_1 \overset{\Pi}{=} q_2$ means that q_1 and q_2 have equivalent external state: $\Pi_\mathrm{e}(q_1) = \Pi_\mathrm{e}(q_2)$.
$N^k(q, q')$ means q' is reachable from q in exactly k steps of N. When N is a function, we write it as n.

The components of a state machine M will be subscripted with "s" for specification and "i" for implementation. We allow *self-loops* in N, e.g. states may transition back to themselves. We assume a machine can always take a step, i.e.,

$$\forall\, q \in Q.\ \exists\, q' \in Q.\ N(q, q').$$

Different correctness statements may require different partitions between internal and external state for the same microprocessor. For example, the externally-visible program counter is sometimes the address of the next instruction to be fetched and sometimes the address of the next instruction to retire.

In verification, the state space of the implementation often needs to be limited to reachable states, or an overapproximation of reachable states. This challenging task is done by finding and proving invariants. Invariants are treated with varying degrees of emphasis in the literature. In our framework we consider the invariants to be encoded in Q, the set of states for the machine.

3 Correctness statements

A well-established definition of correctness is that of *trace containment*: every trace of external observations generated by the implementation can also be generated by the specification. A disadvantage of trace containment is that verifying it can require information about an entire trace. A correctness statement that mitigates this problem is *simulation*: if an implementation state is externally equal to a specification state, then executing one instruction in both the implementation and specification results in states that are externally equal. Simulation can play the role of the induction step in a proof of trace containment; systems that satisfy simulation also satisfy trace containment. Formal verification of sequential microprocessors has generally been done using simulation-style correctness statements. Similar correctness statements are also used in other domains such as cache-coherence protocols [3, 28, 29, 36].

Pipelining and other optimizations increase the gap between the behavior of the implementation and the specification, thus making it more difficult to consider only one step within the implementation and specification traces. Pipelined machines begin executing new instructions before previous ones retire. A superscalar machine may externally appear to do nothing for a number of steps and then, in a single step, update the register file with the results of several instructions. Machines with out-of-order retirement can retire instructions in a different order than the specification.

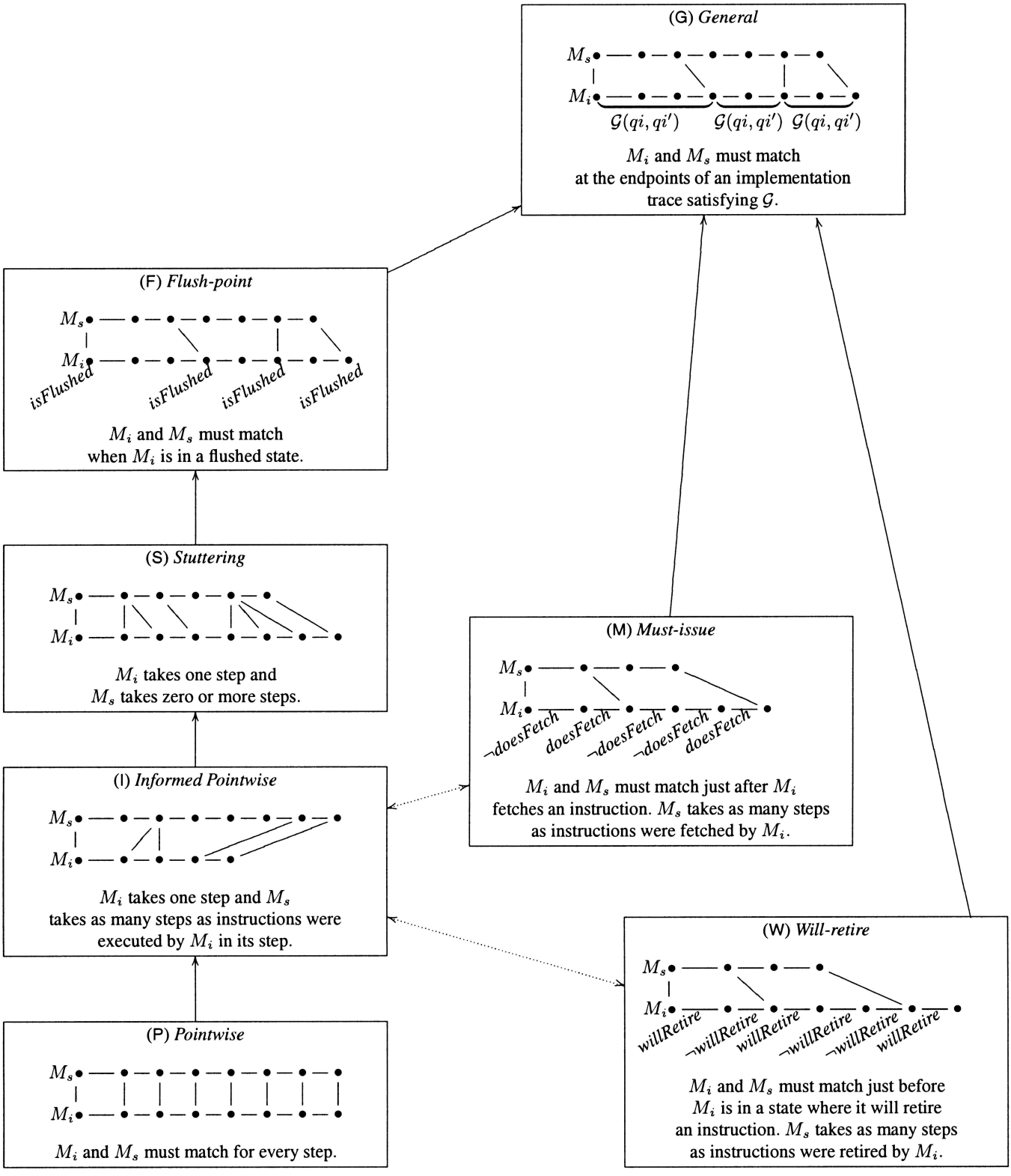
To describe how superscalar verifications use simulation-style correctness statements, we separate the notions of 1) how to *align* the implementation trace against the specification trace to determine which states should match, and 2) what it means for an implementation state to successfully *match* a specification state.

When verifying non-pipelined machines, the traces can be aligned at every step, and two states match if the externally-visible state components are equal. To verify pipelined machines, the alignment often needs to be at looser intervals than every step, or alternatively, external equivalence needs to be replaced by a looser relationship. With superscalar microprocessors, the notions of alignment and matching are necessarily even more complicated. A common alignment technique is to check the implementation only when it is in a *flushed* state (i.e., no in-flight instructions). A common matching relationship is Burch–Dill style flushing [13], which uses an abstraction function to retire all in-flight instructions in the implementation and then checks for external equality with the specification state.

Our framework uses four parameters to characterize a correctness statement: alignment, match, implementation execution, and specification execution. *Alignment* is the method used to align the trace of the implementation with the trace of the specification (Sect. 3.1). *Match* is the relation established between the aligned implementation and specification states (Sect. 3.2). *Implementation execution* and *specification execution* describe the type of state machines used (Sect. 3.3).

3.1 Alignment

Alignment describes which states in the execution traces are tested for matching. Figure 1 illustrates the kinds of alignment that we have found used in microprocessor verification. *Pointwise alignment* (P) is the classic commuting diagram, which compares every step.



In each diagram, the horizontal lines between states are the specification and implementation traces. The vertical lines between states show the where the implementation state must match the specification state.

Fig. 1. Options and partial order for the alignment parameter

Informed-pointwise (I) is a variation of pointwise alignment to handle pipelines that are sometimes unable to fetch an instruction. A microprocessor may be prevented from fetching an instruction in two scenarios. First, the microprocessor might be stalled internally and unable to accept an incoming instruction. Second,

the instruction memory might be unable to provide the microprocessor with an instruction to fetch (e.g. cache miss). Information from the implementation trace about how many instructions have been executed is used to choose how many steps the specification takes before comparing the implementation and specification states. To achieve the effect of informed-pointwise, some authors use a pointwise correctness statement and modify the specification to accept an extra input that is the number of steps to execute. We classify these approaches as informed pointwise.

Stuttering alignment (S) allows the specification to *stutter*, i.e., two or more consecutive implementation states may match the same specification state.

Flush-point alignment (F) says that if there is a trace between flushed implementation states, then there must exist a trace in the specification between a pair of states that match the flushed implementation states. A predicate *isFlushed* indicates when an implementation state is flushed. Instruction set architectures complete the execution of an instruction in a single step; therefore all of their states are flushed.

In *must-issue alignment* (M), the specification takes at least one step, and the implementation takes steps as long as it cannot fetch an instruction, then it takes one more step where it fetches one or more instructions. In this correctness statement only the last step of the trace fetches instructions. The predicate *doesFetch* is true of an implementation step when the implementation fetches one or more instructions. The number of steps taken by the specification is determined by the number of instructions that the implementation fetches. For single-scalar microprocessors, the specification always takes one step.

Will-retire alignment (W) is almost the dual of must-issue alignment. In will-retire alignment, the implementation retires instructions in the first step of the trace. The implementation continues to take steps until it is ready to retire instructions again. The implementation state just before the instructions retire is matched against a specification state. For single-scalar microprocessors, the specification always takes one step. For superscalar microprocessors, the specification may take multiple steps.

Finally, we have included a general case (G) showing that flush-point (F), will retire (W), and must-issue (M) are all instances of a general relation $\mathcal{G}$ on the implementation trace.

In the mathematical formulation of these correctness statements (Sect. 3.5), we provide versions for implementations that can fetch up to w instructions in a single cycle (i.e., w-wide superscalar machines).

The different kinds of alignment form a partial order as illustrated by the arrows in Fig. 1. This order is based on generality where alignments higher in the order are weaker, i.e., implementations that satisfy flush-point alignment may not satisfy pointwise alignment. More precisely, the relationship is implication, with instantiation in the case of the arrows connecting flush-point, will-retire, and must-issue to the general case (G). For example, stuttering correctness implies flush-point for any instance of the predicate *isFlushed*.

The dashed lines between must-issue and informed pointwise alignment, and between will-retire and informed pointwise indicate that the two are related for some match options. For example, with flushing match, must-issue is equivalent to informed pointwise [16]. In addition, assuming the implementation eventually retires an instruction, will-retire with equality match is equivalent to informed-pointwise [15]. These match options are explained in detail in Sect. 3.2.

3.2 Match

Instantiations for the match parameter are relations, $\mathcal{R}$, between an implementation state q_i and specification state q_s that mean "q_i is a correct representation of q_s". Figure 2 shows the matches that we found used in pipelined microprocessor verification. The arrows show the partial order, where definitions lower in the order are instances of higher definitions.

An *other match* (O) is any relation between implementation and specification states. The *abstraction match* (A) uses a function (abs) to map an implementation state to a state that is externally equivalent to the specification state. The *flushing match* (U) is a particular type of abstraction match that uses a flushing function to compute the implementation state that should be externally equivalent a specification state. The *equality match* (E) requires that the implementation and specification states be externally equivalent. The *refinement match* (R) requires that the abstraction function preserve the externally-visible part of the implementation state. Refinement differs from equality because the refinement map is a function, so each implementation state matches exactly one specification state. Words such as "refinement" and "abstraction" have multiple meanings in the literature. The mathematics in Fig. 2 are the precise definitions that we use.

Flushing is often based on iterating a deterministic implementation's next-state function without fetching new instructions. However, other approaches are possible, such as Hosabettu et al.'s completion functions [20].

If the specification does not have any internal state (i.e., all of the state components are externally visible), then equality and refinement both reduce to $\Pi_e(q_i) = q_s$. We call such cases refinement. This relationship is denoted by the dashed line between R and E in Fig. 2.

Figure 2 shows the partial order relationships between these matches. Identifiers such as $\mathcal{R}$ and abs are intended to be substituted with specific relations or abstraction functions in a correctness statement. Other identifiers, such as $flush$ and $=$, have particular meanings. The partial order relationships are implication with some substitution. For example, equality match

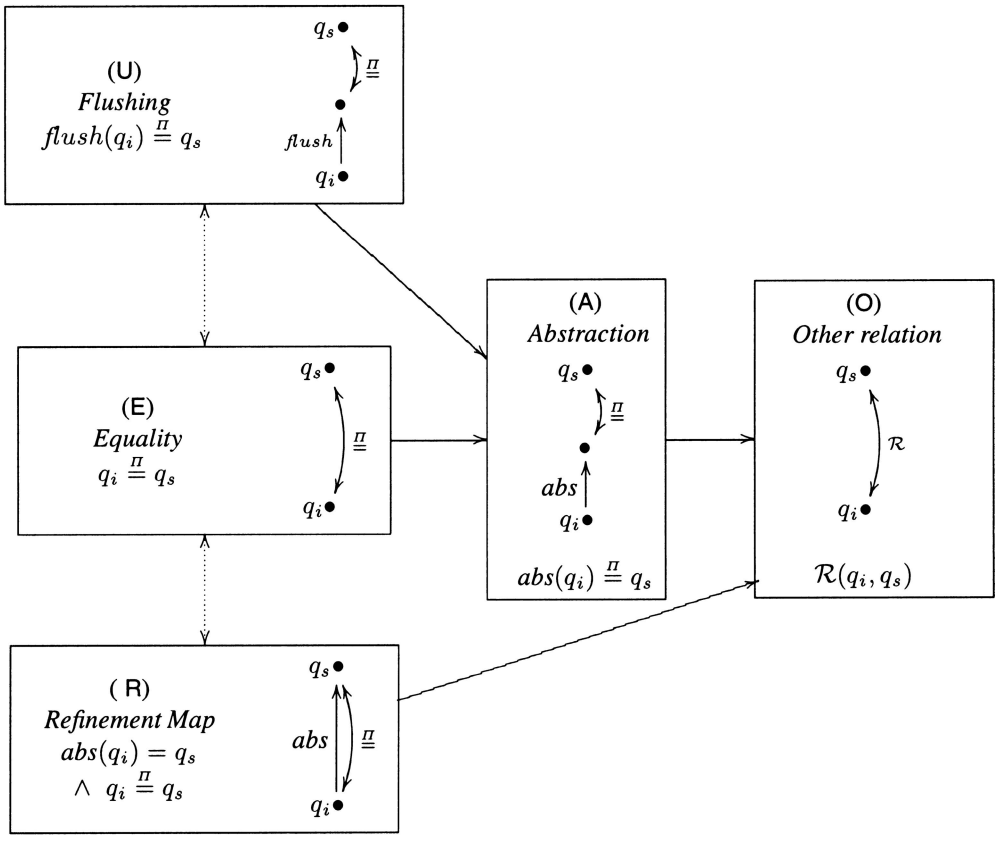


Fig. 2. Options and partial order for the match parameter

and an identity abstraction function implies abstraction ($\mathsf{E} \wedge (abs = \mathrm{id}) \implies \mathsf{A}$).

The dashed line between U and E indicates that flushing is equivalent to the equality match when flush-point alignment is used. That is (FE) is equivalent to (FU), because flushing a flushed state has no effect.

In addition to the options listed here, other options are possible. Pnueli et al. [4, 5, 14, 31] use a matching relation that uses both concretization and abstraction functions. Two states match if concretizing the specification state produces the same result as abstracting the implementation state. In their examples, the concretization functions are identity or projection, so their match specializes to abstraction in our framework.

3.3 Execution

The third and fourth parameters of the framework are the methods for describing the traces of the implementation and specification. In the literature we find both *deterministic* (D) and *non-deterministic* (N) implementations and specifications. In a deterministic machine, the transition relation is instantiated as:

$$N(q, q') \equiv q' = n(q)$$

Implementations are often modeled with non-determinism because of scheduling circuitry. On the other hand, most instruction-set architectures are deterministic, so most specification machines are deterministic. Exceptions include specifications with imprecise exceptions or external interrupts. For our purposes, we consider deterministic machines as instances of non-deterministic machines in the total order for the execution parameters.

3.4 Correctness space

By choosing different options for the parameters, we arrive at a variety of correctness statements. We use four-letter acronyms to describe the combinations of parameter instantiations:

$$\langle alignment \rangle \langle match \rangle \langle impl.\ execution \rangle \langle spec.\ execution \rangle$$

For example, "IUDD" denotes informed-pointwise alignment (I), flushing match (U), and deterministic implementation (D) and specification (D). Figure 3 lists the complete set of acronyms we use for the options of the correctness statements.

The options for each parameter have a total or partial ordering. Together, these orders induce a partial order over correctness statements, which serves to map out the space of correctness statements for microprocessor implementations. Figure 4 shows how we draw the correctness space for implementation and specification execution. Figure 5 is for alignment and match together. All four parameters are

<*alignment*>	<*match*>	<*impl. execution*>	<*spec. execution*>
(M) Must-issue	(O) Other	(N) Non-deterministic	(N) Non-deterministic
(W) Will-retire	(A) Abstraction	(D) Deterministic	(D) Deterministic
(F) Flush-point	(U) Flushing		
(S) Stuttering	(E) Equality		
(I) Informed-pointwise	(R) Refinement Map		
(P) Pointwise			

Fig. 3. Acronyms for options

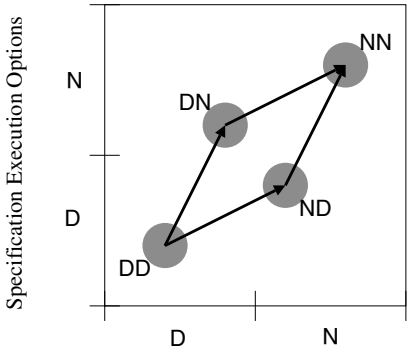


Fig. 4. Partial order for execution options

combined together in Fig. 6, which includes citations to papers that use the listed correctness statements.

In Fig. 5, the shading from MU to IU indicates that with flushing match, must-issue is equivalent to informed pointwise [16]. Similarly, the shading from WR to IR and from WE to IE indicate that for refinement and equality match, will-retire is equivalent to informed pointwise assuming the implementation eventually retires an instruction [15].

The partial order in Fig. 6 is implication with instantiation. For example, IEDN is below SANN in the partial order, because informed-pointwise alignment implies stuttering alignment; equality match implies abstraction match; and deterministic machines are instances of non-deterministic ones. We do not advocate any points in the correctness space over others. The classification serves to highlight the differences and similarities of the different correctness statements.

3.5 Mathematical formulation

In this section we describe the mathematical formulations of correctness statements in the framework. We use $M_i \preccurlyeq_{\mathcal{R}} M_s$ to mean "M_i is correct with respect to M_s via the relation $\mathcal{R}$". All of the correctness statements have the general form of Definition 2.

Definition 2 (General form of correctness statement)

$$(Q_i, Q_i^\circ, N_i) \preccurlyeq_{\mathcal{R}} (Q_s, Q_s^\circ, N_s) \equiv \left[\begin{array}{l} \wedge \forall\, q_i^\circ \in Q_i^\circ.\ \exists\, q_s^\circ \in Q_s^\circ.\ \mathcal{R}(q_i^\circ, q_s^\circ) \\ \wedge \langle \textit{inductive clause} \rangle \end{array}\right]$$

The alignment parameter determines the overall form of the induction clause, while the other parameters have localized effects. For example, from the PONN correctness statement, it is easy to derive variations, such as PANN, PEDN, PRND, etc. In the remainder of this section, we show the induction clauses for the various options of the alignment parameter together with the most-general options for the remaining parameters. That is, we show the "other" relation match with non-deterministic specifications and implementations (∗ONN). We do not show the mathematical definitions of the base clauses, because most are similar to the one shown in Definition 2. We discuss the base cases that vary from this pattern.

The most general combination in the correctness space is where the relevant implementation traces are described by a general relation (GONN, Definition 3). It says that starting from any implementation state that matches a specification state, for all implementation traces that satisfy a general relation $\mathcal{G}$, the specification must step through some number of steps j to reach a state q_s' that matches q_i'.

Definition 3 (General induction clause: GONN)

$$\forall\, q_i, q_i' \in Q_i.\ \forall\, q_s \in Q_s.\ \exists\, q_s' \in Q_s.$$
$$\left[\begin{array}{l} \wedge (\exists\, k.\ N_i^k(q_i, q_i')) \\ \wedge \mathcal{G}(q_i, q_i') \\ \wedge \mathcal{R}(q_i, q_s) \end{array}\right] \Longrightarrow \left[\begin{array}{l} \wedge \exists\, j.\ N_s^j(q_s, q_s') \\ \wedge \mathcal{R}(q_i', q_s') \end{array}\right]$$

Definition 4 is for flush-point alignment with a general match and non-deterministic machines (FONN). It says that if the implementation is in a flushed state q_i and can step through some number of steps k to another flushed state q_i', then all specification states q_s that match q_i (via $\mathcal{R}$) must step through some number of steps j to some state q_s' that matches q_i'.

Definition 4 (Flush-point induction clause: FONN)

$$\forall\, q_i, q_i' \in Q_i.\ \forall\, q_s \in Q_s.\ \exists\, q_s' \in Q_s.$$
$$\left[\begin{array}{l} \wedge \mathit{isFlushed}(q_i) \\ \wedge \exists\, k.\ N_i^k(q_i, q_i') \\ \wedge \mathit{isFlushed}(q_i') \\ \wedge \mathcal{R}(q_i, q_s) \end{array}\right] \Longrightarrow \left[\begin{array}{l} \wedge \exists\, j.\ N_s^j(q_s, q_s') \\ \wedge \mathcal{R}(q_i', q_s') \end{array}\right]$$

The base case for flush-point alignment conforms to the pattern in Definition 2, because we assume the initial state of the implementation is a state resulting from resetting the implementation. Such a state would be flushed.

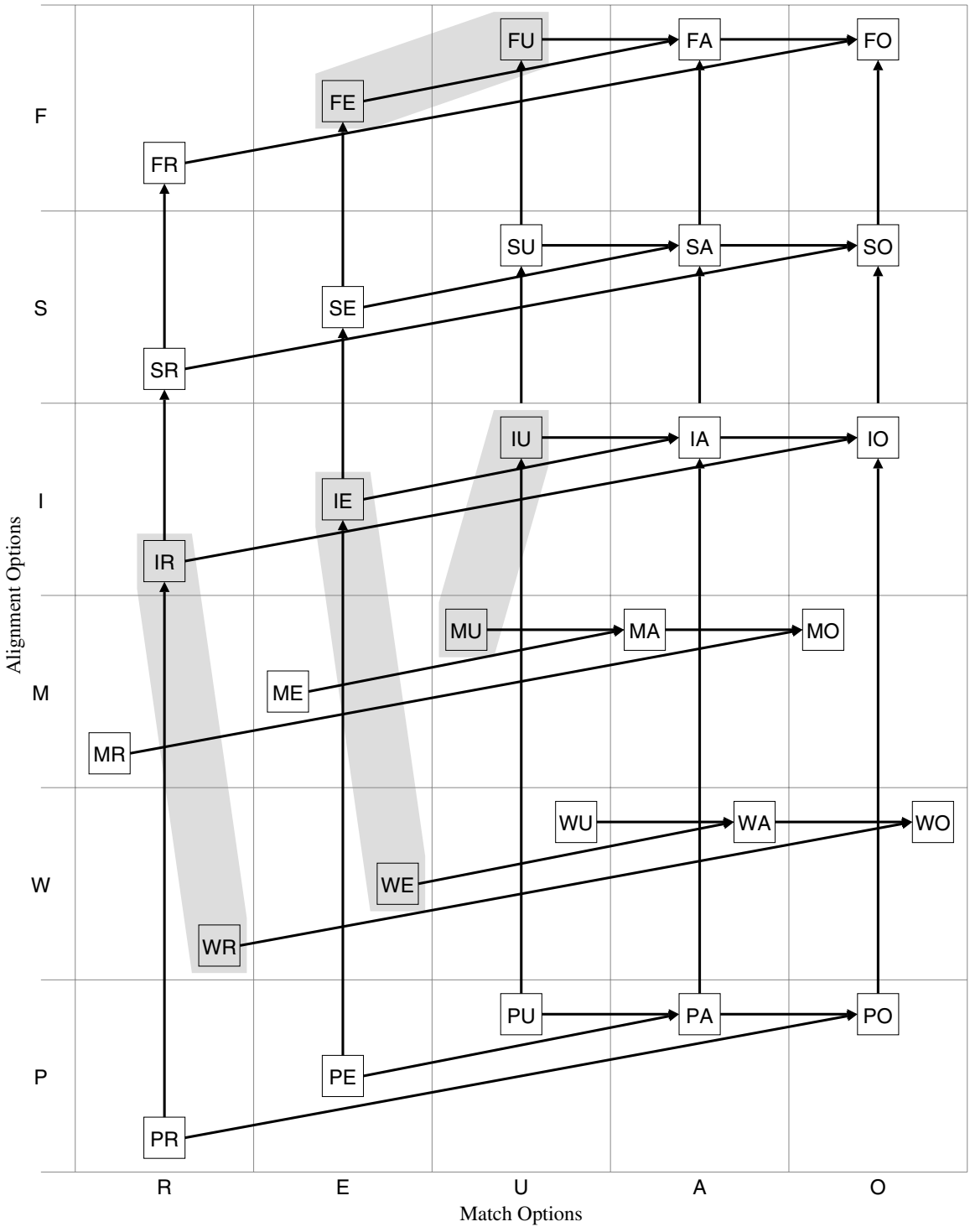


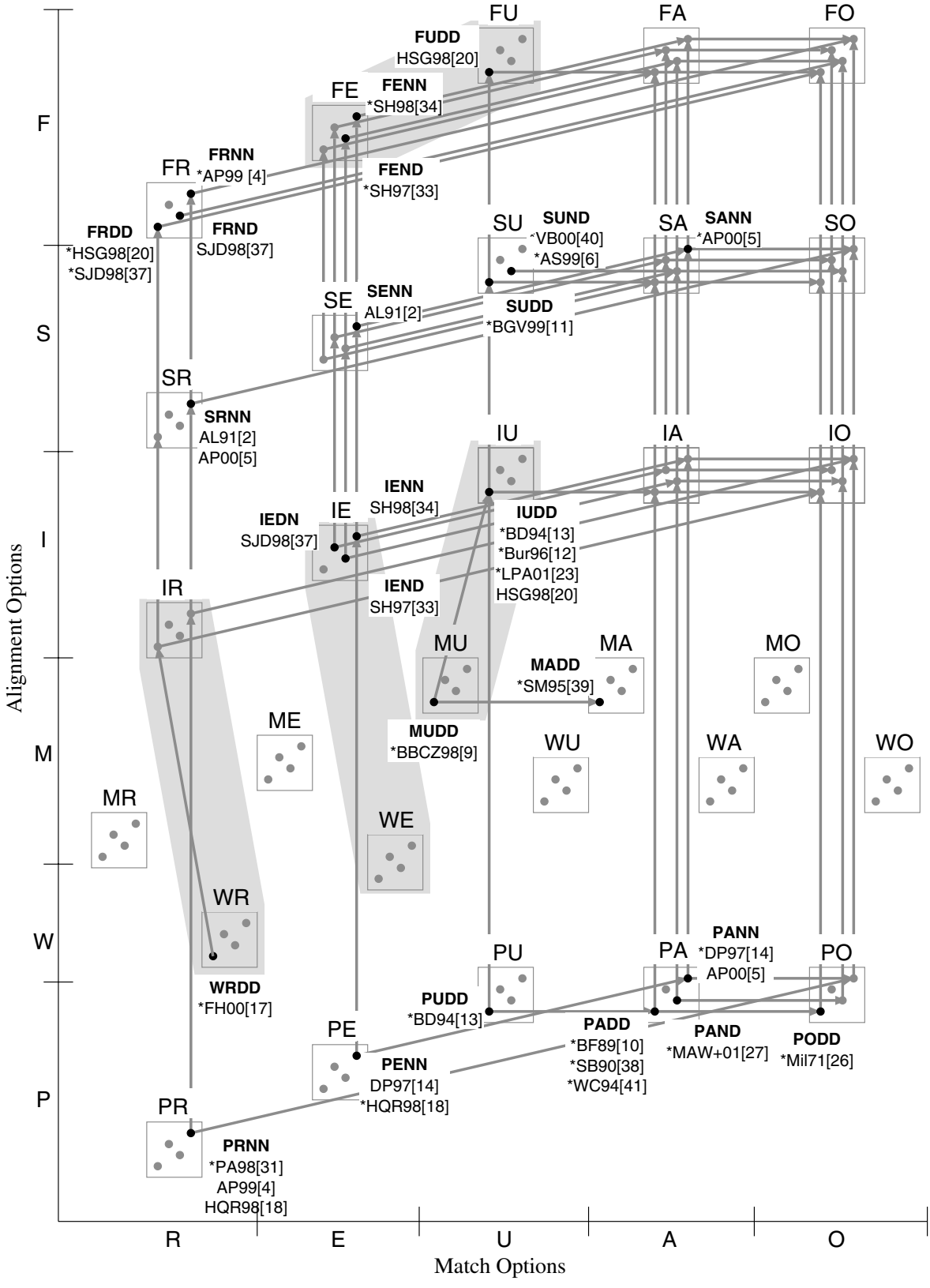
Fig. 5. Partial order for alignment and match options

Stuttering alignment (Definition 5) results in a simpler correctness statement than flushpoint alignment, because it considers only one step of the implementation and requires no special predicates. The specification is allowed to *stutter*, e.g., consecutive implementation states may align with the same specification state. The specification may take up to w steps where w is the width of the machine.

Definition 5 (Stuttering induction clause: SONN)

$$\forall q_i, q_i' \in Q_i.\ \forall q_s \in Q_s.\ \exists q_s' \in Q_s.$$
$$\begin{bmatrix} N_i(q_i, q_i') \\ \wedge \mathcal{R}(q_i, q_s) \end{bmatrix} \implies \begin{bmatrix} \exists\, j \leq w.\ N_s^j(q_s, q_s') \\ \wedge \mathcal{R}(q_i', q_s') \end{bmatrix}$$

In pointwise alignment (Definition 6) the specification always takes one step. Pointwise alignment requires that the implementation is always able to execute an instruc-



Each point is annotated with citations for the works that use the particular correctness statements. Citations prefixed with ∗ denote top-level correctness statements; others are used as intermediate correctness statements during the proofs. Sect. 4 provides further explanation.

Fig. 6. Space of correctness statements

tion. For a superscalar microprocessor of width w, this would require that it always be able execute w instructions. As this is clearly unrealistic, we have not generalized pointwise alignment for superscalar machines.

Definition 6 (Pointwise induction clause: PONN)

$$\forall q_i, q_i' \in Q_i.\ \forall q_s \in Q_s.\ \exists q_s' \in Q_s.$$
$$\left[\begin{array}{l}N_i(q_i, q_i')\\ \wedge \mathcal{R}(q_i, q_s)\end{array}\right] \implies \left[\begin{array}{l}N_s(q_s, q_s')\\ \wedge \mathcal{R}(q_i', q_s')\end{array}\right]$$

Informed-pointwise alignment (Definition 7) is very similar to pointwise alignment. The difference is that the number of steps taken by the specification is determined by the implementation trace using the function *numInstr*. This is done to accommodate situations when the implementation does not execute an instruction, and, in the case of superscalar machines, does not execute the maximum number of instructions possible. The number of steps taken by the specification can never exceed the execution width w of the implementation.

Definition 7
(Informed-pointwise ind. clause: IONN)

$$\forall q_i, q_i' \in Q_i.\ \forall q_s \in Q_s.\ \exists q_s' \in Q_s.$$
$$\text{let } j = \mathit{numInstr}(q_i, q_i') \text{ in}$$
$$\left[\begin{array}{l}N_i(q_i, q_i')\\ \wedge \mathcal{R}(q_i, q_s)\end{array}\right] \implies \left[\begin{array}{l}N_s^j(q_s, q_s')\\ \wedge \mathcal{R}(q_i', q_s')\end{array}\right]$$

The function *numInstr* is, in practice, instantiated with either *numInstrFetched* or *numInstrRetired*. In general, *numInstrFetched* is used with flushing abstraction.

In must-issue alignment (Definition 8), the implementation takes as many steps (k) as are necessary to reach a state where it can fetch an instruction; it then takes one more step from q_i^k to q_i^{k+1} where it fetches one or more instructions. The number of steps the specification takes depends on how many instructions are fetched in the last implementation step.

Definition 8 (Must-issue induction clause: MONN)

$$\forall q_i^0, q_i^1, \ldots, q_i^{k+1} \in Q_i.\ \forall q_s \in Q_s.\ \exists q_s' \in Q_s.$$
$$\text{let } f = \mathit{numInstrFetched}(q_i^k, q_i^{k+1}) \text{ in}$$
$$\left[\begin{array}{l}\left[\begin{array}{l}\forall j < k.\\ \wedge N_i(q_i^j, q_i^{j+1})\\ \wedge \neg \mathit{doesFetch}(q_i^j, q_i^{j+1})\end{array}\right]\\ \wedge N_i(q_i^k, q_i^{k+1})\\ \wedge \mathit{doesFetch}(q_i^k, q_i^{k+1})\\ \wedge \mathcal{R}(q_i^0, q_s)\end{array}\right] \implies \left[\begin{array}{l}N_s^f(q_s, q_s')\\ \wedge \mathcal{R}(q_i^{k+1}, q_s')\end{array}\right]$$

Similar in form to must-issue alignment is will-retire alignment (Definition 9). In will-retire alignment, the first step of the implementation trace retires one or more instructions. The number of steps the specification takes depends on how many instructions are retired in the first implementation step. This number is returned by the function *numInstrRetired*. The implementation runs from the step where it retires one or more instructions to a state where will next retire instruction(s).

Definition 9 (Will-retire induction clause: WONN)

$$\forall q_i^0, q_i^1, \ldots, q_i^k \in Q_i.\ \forall q_s \in Q_s.\ \exists q_s' \in Q_s.$$
$$\text{let } r = \mathit{numInstrRetired}(q_i^0, q_i^1) \text{ in}$$
$$\left[\begin{array}{l}N_i(q_i^0, q_i^1)\\ \wedge \mathit{willRetire}(q_i^0, q_i^1)\\ \wedge \left[\begin{array}{l}\forall j \in 1 \ldots k-1.\\ \wedge N_i(q_i^j, q_i^{j+1})\\ \wedge \neg \mathit{willRetire}(q_i^j, q_i^{j+1})\end{array}\right]\\ \wedge \left[\begin{array}{l}\exists q_i.\\ \wedge N_i(q_i^k, q_i)\\ \wedge \mathit{willRetire}(q_i^k, q_i)\end{array}\right]\\ \wedge \mathcal{R}(q_i^0, q_s)\end{array}\right] \implies \left[\begin{array}{l}N_s^r(q_s, q_s')\\ \wedge \mathcal{R}(q_i^k, q_s')\end{array}\right]$$

The base case for will-retire alignment is more complicated than any of the other base cases. The inductive step of will-retire alignment requires that the implementation retires an instruction. The base case of a correctness statement must connect the initial state (e.g., just after reset is deasserted) to the inductive step. For will-retire alignment, this requires that the base case steps the implementation from the initial state until just before it retires an instruction.

Section 4 describes other correctness statements as points in the correctness space by instantiating $\mathcal{R}$ and the next-state relations in each of the above four definitions. Instantiating next-state relations with functions removes the need for some quantified variables.

3.6 Limitations

As with most formal verification approaches, there are certain degenerate configurations that can lead to vacuous verification results. The correctness framework in and of itself cannot prevent all of these possibilities. However, all of the vacuous examples we have found rely on degenerate matching relations. For example, a matching relation that includes all implementation and specification state pairings will result in a vacuously-true correctness statement, as any pair of consecutive implementation states can be related to any pair of consecutive specification states. Another vacuous statement results from using stuttering alignment with a degenerate abstraction function that maps all implementation states to the same specification state.

4 Literature survey

In this section, we show how a variety of correctness statements for out-of-order and superscalar microprocessors are points in the correctness space defined by the framework. To set this work in an historical context, we give an overview of verification work with commuting

diagrams and abstraction functions. This overview begins with Milner's definition of *simulation* in 1971 and ends with manually constructed abstraction functions for single-scalar pipelined microprocessors.

In the discussion of microprocessor verification, we describe thirty-seven correctness statements from twenty-nine papers. While the phrase "we use the correctness statement of Burch and Dill [13]" appears in many papers, detailed examination reveals that this is often an approximation. For conciseness, we focus on the inductive clauses of the correctness statements.

4.1 Historical perspective

Milner's [26] work in software verification led to a definition of simulation that is *pointwise alignment* of a general relation between a deterministic implementation and a deterministic specification (PODD). Milner does not include a base clause. PODD is derived from Definition 6 (*pointwise alignment*) by substituting next-state functions (n_s and n_i) for the next-state relations (N_s and N_i).

Definition 10 (Milner's *simulation*: PODD)
$\forall\, q_i \in Q_i.\, \forall\, q_s \in Q_s.\ \mathcal{R}(q_i, q_s) \implies \mathcal{R}(n_i(q_i),\ n_s(q_s))$

Abadi and Lamport [2] define *refinement maps*; in our parlance this is *stuttering refinement* between nondeterministic machines (SRNN). They use refinement as a verification strategy to prove *stuttering equivalence* (SENN), which they call *implements*. Both SRNN and SENN are derived from Definition 5 (SONN) for a one-wide machine ($w = 1$). For SRNN, refinement, $(abs(q_i) = q_s) \wedge (q_i \stackrel{\Pi}{=} q_s)$, is substituted for the match $\mathcal{R}$, which results in Definition 11:

Definition 11
(Abadi and Lamport's *refines*: SRNN)
$\forall\, q_i, q_i' \in Q_i.$

$$\begin{bmatrix} \wedge^{N_i(q_i, q_i')} \\ q_i \stackrel{\Pi}{=} abs(q_i) \end{bmatrix} \implies \begin{bmatrix} \begin{bmatrix} \vee^{N_s(abs(q_i), abs(q_i'))} \\ abs(q_i) = abs(q_i') \end{bmatrix} \\ \wedge q_i' \stackrel{\Pi}{=} abs(q_i') \end{bmatrix}$$

Their main result is that if SENN holds, then it is possible to construct an intermediate model from an implementation using history and prophecy variables such that the intermediate model will satisfy SRNN with the specification.

Several verifications of single-scalar pipelines use correctness statements that are relevant to our paper. Bose and Fischer [10] used PADD in the verification of a pipelined stack. In the first published verification of a pipelined microprocessor, Srivas and Bickford [38] proved PADD between their implementations and specification. Later, Windley and Coe [41] also verified PADD. Srivas and Miller [39] proved MADD between a pipelined microprocessor at the instruction and micro-instruction levels of abstraction. All of the abstraction functions in these efforts were manually constructed.

4.2 Burch and Dill (flushing)

Manual construction of an implementation abstraction function, as in the works cited in the previous section, is a difficult and tedious process. *Flushing*, a technique for deriving the implementation abstraction function automatically, was introduced in a seminal paper by Burch and Dill [13]. To abstract the implementation state, they force the implementation to send bubbles down the pipeline. This flushes all in-flight instructions out of the pipeline and into architectural state. This behaviour is common in microprocessors that have to handle delays in fetching instructions from memory, such an instruction cache miss, but might not be a part of all implementations. In [13], they verify PUDD (*pointwise flushing*) for a simple 3-stage pipelined ALU.

In general, pipelines do not execute instructions every clock cycle. This complicates alignment with a flushing abstraction function. If the implementation cannot fetch an instruction to take a productive step, flushing the states before and after the implementation step will result in the same state. Various authors have different ways to handle this situation. These correspond to different points in the framework.

Burch and Dill [13] used *informed-pointwise alignment*, denoted "I". The correctness statements with *informed-pointwise alignment* use stall information from the implementation to determine how many steps the specification should take. In this alignment option, we also include approaches that pass the stall information to the specification. Burch and Dill also use flushing to verify IUDD between an instruction set architecture specification and a single-scalar 5-stage pipeline implementation. In this case, *numInstrFetched* is equal to zero when the pipeline is stalled and one when it fetches an instruction. Burch later generalized this idea to superscalar microprocessors [12], by making *numInstrFetched* a function that indicates to the specification how many instructions the implementation fetched. In our framework this is IUDD with $w > 1$. Lahiri et al. used Burch's approach to verify a pipeline description of the 7-stage, dual-issue Motorola M*CORE^{TM} processor [23]. Other approaches that handle this alignment problem with flushing abstraction can be found in Sects. 4.3, 4.4, 4.6, and 4.9.

4.3 Bryant et al.

Bryant et al. [11] verified *stuttering flush* (SUDD) between a superscalar processor and an instruction set architecture. The SUDD correctness criteria is similar to the previously-cited approaches that use *informed-pointwise alignment* with *flushing* (IU). However, instead of using the number of instructions fetched by the implementa-

tion to determine the number of specification steps, they explicitly state a disjunction of equalities between specification and implementation states. The disjuncts cover the different cases for the number of steps the specification can take: 0 to w, where w is the fetch width of the machine In a later paper, Velev and Bryant [40] verify SUND. The nondeterminism is added to the implementation description to facilitate modeling of variable-latency instructions.

4.4 Hosabettu et al. (completion functions)

Hosabettu, Srivas, and Gopalakrishnan [20, 21] prove that a deterministic out-of-order implementation satisfies *flush-point refinement* with a deterministic specification where the match is projection (Definition 12).

Definition 12 (Flush-point refinement with projection: an instance of FRDD)

$$\forall\, q_i \in Q_i.\ \forall\, k.$$
$$\begin{bmatrix} isFlushed(q_i) \\ \wedge\, isFlushed(n_i^k(q_i)) \end{bmatrix} \Longrightarrow \begin{bmatrix} \exists\, j. n_s^j(\Pi_e(q_i)) \\ = \\ \Pi_e(n_i^k(q_i)) \end{bmatrix}$$

Because their verification is completely within a theorem prover, they are able to use underspecified next-state functions (rather than relations) for their scheduler. They prove FRDD in three steps. They prove *informed-pointwise flushing* (IUDD) and then apply induction to prove *flush-point flushing* (FUDD). They go from FUDD to FRDD by proving that the flushing abstraction of a flushed state is equivalent to projection.

Their flushing abstraction function is constructed from *completion functions*. Completion functions describe the effect on the observable state of completing the execution of each in-flight instruction. Completion functions are written manually and composed pipe stage by pipe stage working backwards through the implementation to create a flushing function.

Hosabettu et al. use *synchronization functions* to tell the specification how many steps it should take to match the implementation. In our framework, the synchronization functions appears as the *numInstrFetched* parameter to informed-pointwise correctness statements. Hosabettu et al. [19] also use the same correctness statement to verify an implementation with speculative execution and precise exceptions.

4.5 Skakkebæk et al. (incremental flushing)

Skakkebæk et al. [37] verify that a deterministic implementation with in-order retirement satisfies *flush-point refinement* with a deterministic specification (FRDD). They build a non-deterministic intermediate model that computes the result of each instruction when it enters the machine and queues the result for later retirement. This intermediate model has hidden state relative to the implementation. The verification of the implementation against the intermediate model shows *informed-pointwise equality* (IEDN). The verification of the intermediate model against the specification establishes *flush-point refinement* (FRND) by incrementally decomposing a monolithic flushing abstraction function into a set of simpler flushing steps. In [22], they use a non-deterministic intermediate model with an abstracted scheduler that provides fine-grained control over instruction progress. This reduces the amount of manual abstraction required by strengthening the simpler flushing steps.

4.6 Berezin et al. (must-issue)

Berezin et al. [9] prove *must-issue alignment* with the *flushing match* (MUDD) for a processor with out-of-order retirement (Definition 13). The model is deterministic but some of the scheduling is left underspecified. They introduce intermediate models of the implementation and specification that are optimized for model-checking efficiency. They prove MUDD between the intermediate implementation and intermediate specification. When used with a deterministic implementation, the predicate *doesFetch* only depends on the implementation state at the beginning of the step. Their verification is for a single-scalar implementation so we omit the function *numInstr*.

Definition 13 (Must-issue flushing MUDD)

$$\forall q_i \in Q_i.\ \forall\, k.$$
$$\begin{bmatrix} \begin{bmatrix} \forall\, j < k-1. \\ \neg doesFetch(n_i^j(q_i)) \end{bmatrix} \\ \wedge\, doesFetch(n_i^{k-1}(q_i)) \end{bmatrix} \Longrightarrow \begin{bmatrix} n_s(\Pi_e(flush(q_i))) \\ = \\ \Pi_e(flush(n_i^k(q_i))) \end{bmatrix}$$

4.7 Sawada and Hunt (trace tables)

Sawada and Hunt [33] verified that a non-deterministic implementation with out-of-order retirement satisfies *flush-point equality* with a deterministic specification (FEND, Definition 14). FEND results from substituting the equality match ($q_i \stackrel{\Pi}{=} q_s$) for $\mathcal{R}$ and a next-state function n_s for the next-state relation in Definition 4 (FONN).

Definition 14 (Flush-point equality: FEND)

$$\forall\, q_i, q_i' \in Q_i.\ \forall\, q_s \in Q_s.$$
$$\begin{bmatrix} isFlushed(q_i) \\ \wedge [\exists\, k.\ N_i^k(q_i, q_i')] \\ \wedge\, isFlushed(q_i') \\ \wedge\ q_i \stackrel{\Pi}{=} q_s \end{bmatrix} \Longrightarrow \left[\exists\, j.\ q_i' \stackrel{\Pi}{=} n_s^j(q_s)\right]$$

In later work [34, 35], they enhanced their implementation to support in-order retirement, external interrupts, and precise exceptions. The inclusion of interrupts led them to add non-determinism to their specification, to

account for the problem of predicting how many instructions the implementation will have completed when an interrupt occurs. They kept *flush-point equality* as their alignment and match criteria, making their correctness statement FENN.

Throughout their work, the verification strategy is to build an intermediate model with history variables. The intermediate model contains an unbounded table, called a MAETT, with one entry for each issued instruction. In their first work [33], they prove *informed-pointwise equality* (IEND) between the implementation and intermediate model and *flush-point equality* (FEND) between the intermediate model and specification, which together imply FEND. Similarly for their second model, they prove IENN and FENN respectively, to conclude FENN.

4.8 Pnueli et al. (variations on refinement)

Four works by the authors Damm, Pnueli, and Arons use a wide range of correctness statements and implementations. Damm and Pnueli [14] prove *pointwise abstraction* (PANN) for an implementation with out-of-order retirement. Their non-deterministic specification generates all possible traces of a program that obey data-dependencies, which allows them to use pointwise alignment. They introduce an intermediate model with auxiliary variables and prove *pointwise equality* (PENN) between the implementation and the intermediate model, and PANN between the intermediate model and the specification. For PANN, their abstraction projects the current implementation state if all instructions have retired. Otherwise, it returns the initial implementation state.

Arons and Pnueli [4] prove *flush-point refinement* (FRNN) for an implementation with out-of-order retirement. The specification can self-loop at every state, but is otherwise deterministic. They use an intermediate model with history variables and prove that whenever the implementation is flushed, the history variables match the implementation (FRNN). They verify that the intermediate model satisfies *pointwise refinement* (PRNN) with the specification. Subsequently, Pnueli and Arons change their synchronization point from instruction issue to instruction retirement, which allows them to tighten their top-level correctness statement to be *pointwise refinement* (PRNN) for an implementation with in-order retirement [31].

Arons and Pnueli [5] verify *stuttering abstraction* (SANN) for a machine with speculative execution, precise exceptions, and in-order retirement. Their abstraction computes the abstract program counter based on the contents of the reorder buffer. They perform two different verifications, one based on induction over the size of the reorder buffer and one using abstraction functions. In the inductive proof, they use three intermediate models to prove SANN: *stuttering refinement* (SRNN), relying on the result of [2]; *pointwise abstraction* (PANN); and SANN.

4.9 Arvind and Shen

Arvind and Shen [6] use term rewriting systems to model an out-of-order processor and its instruction set architecture. Their specification is deterministic. Their implementation consists of a set of rewrite rules, where multiple rewrite rules are required to complete an implementation step, or clock cycle. Because the implementation could have multiple rules satisfied in the term rewriting system at a given time, their implementation is non-deterministic.

Their top-level correctness statement is for any sequence of rewrite rules, which corresponds to any length of implementation trace, starting from any implementation state. A specification trace of zero or more rewriting rules (zero or more specification steps) must exist for each implementation trace. If the implementation stalls or doesn't progress according to the specification's view, the specification could take zero steps. Therefore, we characterize their alignment as *stuttering*. However, their correctness statement is at the granularity of rewrite rules, and one rewrite rule may accomplish less than a complete implementation step. Therefore, their correctness statement requires the implementation and specification to match at even smaller intervals than a single step.

Their match criteria is *flushing*, which they refer to as *draining*. Like many other approaches, an implementation state is flushed and then the programmer-visible state is projected for an equivalence check with the specification. Because the implementation could have multiple rules satisfied in the term rewriting system at a given time, their implementation is non-deterministic. Thus, they verify SUND between the out-of-order implementation and the instruction set architecture.

Using term rewriting systems, it is possible that the flushing process is non-deterministic, i.e., flushing might not be a function. In these cases, their match would be O.

Their verification system allows for the possibility of using a variety of different abstraction mechanisms. For example, rather than committing partially-executed instructions, they could roll back the effects of the instructions as if they had never been fetched. This rolling back of execution could be used to synchronize their implementation with the specification at instruction retirement, rather than instruction fetch.

Arvind and Shen also show a liveness property: that the implementation includes all possible behaviors of the specification.

4.10 Henzinger et al. (assume-guarantee)

Henzinger et al. [18, 32] use a top-level correctness statement of *pointwise equality* (PENN), which they call *trace containment*, to prove the correctness of an out-of-order retirement processor where both the specification and implementation may have internal state. Their specification includes a non-deterministic stall signal and the schedul-

ing in their implementation is non-deterministic. They construct abstraction and witness modules to bridge the gap between the specification and implementation. Using assume-guarantee reasoning, they reduce the problem to smaller proof obligations where the specification has no internal state. In these cases, which they call *projection refinement*, they prove *pointwise refinement* (PRNN).

4.11 Fox and Harman

Fox and Harman [17] are unique in using *will-retire alignment*, which aligns the specification with the implementation whenever an instruction is about to retire. Their match is refinement, where their refinement function is projection. They synchronize at instruction retirement, rather than the more common synchronization point of instruction fetch. The program counter in the externally-visible state is for the instruction about to retire, instead of the one about to be fetched. Both their implementation and specification are deterministic. This gives them a correctness criteria of WRDD (Definition 15). For a deterministic implementation, the function *numInstrRetired* and the predicate *willRetire* depend only on the implementation state at the beginning of the trace. Their verification techniques use algebraic rewriting. They have used both single-scalar and superscalar versions of their correctness statement.

Definition 15 (Will-retire with refinement: WRDD)

$$\forall\, q_i \in Q_i.\ \forall\, q_s \in Q_s.\ \forall\, k.$$
$$\text{let } r = \mathit{numInstrRetired}(q_i) \text{ in}$$
$$\begin{bmatrix} \mathit{willRetire}(q_i) \\ \wedge \begin{bmatrix} \forall j \in 1 \ldots k-1. \\ \quad \neg \mathit{willRetire}(n_i^j(q_i)) \end{bmatrix} \\ \mathit{willRetire}(n_i^k(q_i)) \end{bmatrix} \Longrightarrow \begin{bmatrix} n_s^r(\Pi_e(q_i)) \\ = \\ \Pi_e(n_i^k(q_i)) \end{bmatrix}$$

The formulation of alignment in their correctness statement is more general than they have actually used. Their correctness statement could be used for pointwise, informed pointwise, must-issue, and flushpoint alignment. Their verification strategy relies on the use of functions. This prevents them from using stuttering alignment, matches other than refinement, and non-deterministic implementations or specifications.

4.12 Austin, Mneimneh et al.

Austin [7] introduced a novel approach to microprocessor architecture with a dual-core implementation: an aggressive performance-oriented core is checked in real-time by a much simpler checker core. If the checker and performance cores are interfaced appropriately, the checker will discover any errors made by the performance core. As a result, formal verification of the checker core is sufficient to show overall correctness of the complete implementation.

Mneimneh et al. [27] verified one mode of the simple checker core against an instruction set architecture. The checker core has two modes: check and recovery. In recovery mode, which they verified, only one instruction is allowed in the pipeline at a time. We characterize this as *pointwise alignment* (PRDD), where the implementation takes internal steps while the instruction moves through the machine. Based on their existing approach, a logical extension of their work would prove IUDD of the checker core in check mode, which supports normal pipelined operation of the checker.

4.13 Related correctness paradigm

Manolios [24] defines correctness based on *well-founded bisimulation*. He allows both the specification and implementation to be non-deterministic and to stutter, but also includes a liveness property that guarantees that they will stutter for only finitely many steps. This approach has not yet been applied to out-of-order implementations. If we excise the liveness criteria from his correctness statement, his work can be characterized as verifying that the implementation satisfies *stuttering equivalence* against the specification and that the specification satisfies the same property with the implementation.

5 Discussion

We have presented a framework for describing microprocessor correctness statements that enables us to compare existing correctness statements and to highlight the differences among them. Our classification is meant as a stepping stone towards understanding the links between an implementation's features, the desired "strength" of correctness statement, and the verification techniques. Indeed, the framework has led us to a number of observations that we now discuss.

Machines with out-of-order retirement are difficult to align with specifications, because they can reach states that are not possible when executing instructions sequentially. One possibility is to use equality match, a deterministic specification, and *flush-point alignment*. Two other approaches allow the use of *informed-pointwise alignment*: an abstraction function that retires all in-flight instructions (e.g., completion functions [20]) or a non-deterministic specification that allows different retirement orderings [14]. In theory, the abstraction function could be simple flushing [13]. In practice, applying simple flushing to an out-of-order machine is infeasible due to capacity limits in verification.

Sawada and Hunt [33] have verified an out-of-order implementation two different ways: using *flush-point equality* (Definition 14) and Burch–Dill style *informed-pointwise abstraction* (Definition 7). They found *flush-point equality* to be significantly easier. Hosabettu [20] first verifies

pointwise flushing and then concludes *flushpoint refinement* because flushing a flushed state has no effect. From this and other anecdotal evidence, we postulate that *flush-point equality* is a verification convenience, i.e., realistic machines that satisfy *flush-point equality* will also satisfy *informed-pointwise abstraction* or *stuttering abstraction*. In the case of machines without external interrupts, a flushing-style abstraction function should suffice, while a machine with interrupts would require a more sophisticated abstraction function to keep the interrupt trace aligned between the specification and implementation.

Stalls complicate the alignment of the implementation and specification. Pnueli and Arons [31] use *pointwise refinement* (PRNN) with a specification that self-loops when no instruction retires. Others use *informed-pointwise* with the *flushing abstraction* where the implementation is flushed and the specification self-loops when no instruction is issued. A recent trend is to use *flush-point equality* [33, 34] or *flush-point refinement* [4, 20, 37], where the implementation and specification are compared only when the implementation is in a flushed state.

Verifying machines with exceptions complicates the instantiation of the match parameter. Most approaches in the literature synchronize the implementation and specification machines at instruction issue. However, Damm and Pnueli [14] and Arons and Pnueli [5] synchronize at retirement, an approach that makes it easier to handle exceptions. The synchronization point is encapsulated in the definition of the match and alignment parameters and is not distinguished by our framework.

In Fig. 6, almost all of the intermediate correctness statements lead to the top-level correctness statements by tracing along the edges in the graph. The two exceptions are *completion functions* [20] and *incremental flushing* [37], whose use of mechanized theorem proving enables these more complicated verification strategies.

It is natural to ask: when starting a verification, which correctness statement should I try to prove? In practice, the choice of correctness statement depends on the complexity of the microarchitectural model in question and the verification technology available. This issue deserves further study, but we can make a few preliminary observations. If the implementation does not always execute an instruction, then *informed-pointwise* is chosen over *pointwise* alignment. If the implementation can execute instructions out-of-order, then *flush-point alignment* is often chosen, because for tighter alignments the match becomes overly complicated. Finally, if the verification technology of choice is based on symbolic simulation, the implementation is usually deterministic.

In related work, we have formalized the framework in a theorem prover and mechanically verified the partial order between correctness statements [15, 16]. Through this mechanization process, we have discovered and verified relationships among more specific points in the framework. For example, we have proved that *must-issue alignment* with a *flushing match* implies *flush-point alignment* with an *equality match*. Our goal is to link this mechanized, reusable theory with particular verification strategies such as completion functions [20] or the MAETT [33].

Our framework is not an end in itself. Rather, it should be used as a foundation for further investigation and a deeper understanding of developments in the formal verification of high-level models of microprocessors. There are options for the framework's parameters that we have not enumerated, and we anticipate that some of these will find useful application. For example, as other approaches besides Sawada and Hunt [34] begin to include external interrupts, we anticipate that additional points in the correctness space will be explored. It remains to be determined what the framework indicates about the relative quality of correctness criteria. It would also be fruitful to explore the potential of using the framework to predict the difficulty of different verification approaches.

Acknowledgements. We thank Andrew Martin of Motorola, Meng Lou of the University of Waterloo, Ken McMillan of Cadence, and Anthony Fox of the University of Cambridge for helpful discussions about this work. We also thank the anonymous reviewers for their helpful comments. The first and third authors are supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC). The first author is also supported in part by a grant from Intel Corporation.

Note added in proof
In subsequent work, the framework described in this paper is called Microbox.

References

1. Aagaard, M.D., Cook, B., Day, N.A., Jones, R.B.: A framework for microprocessor correctness statements. In: Margaria, T., Melham, T., (eds.), IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME), Lecture Notes in Computer Science, vol. 2144. Springer, Berlin Heidelberg New York, 2001, pp. 433–448
2. Abadi, M., Lamport, L.: The existence of refinement mappings. J Theoret Comput Sci 2(82):253–284, 1991
3. Arons, T.: Using timestamping and history variables to verify sequential consistency. In: Berry, G., Comon, H., Finkel, A., (eds.), Workshop on Computer-Aided Verification (CAV), Lecture Notes in Computer Science, vol. 2102. Springer, Berlin Heidelberg New York, 2001, pp. 423–435
4. Arons, T., Pnueli, A.: Verifying Tomasulo's algorithm by refinement. In: International Conference on VLSI Design, pp. 92–99, 1999
5. Arons, T., Pnueli, A.: A comparison of two verification methods for speculative instruction execution with exceptions. In: Graf, S., Schwartzbach, M., (eds.), Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Lecture Notes in Computer Science, vol. 1785. Springer, Berlin Heidelberg New York, 2000, pp. 487–502
6. Arvind, and Shen, X.: Using term rewriting systems to design and verify processors. IEEE Micro 19(3):36–46, 1999
7. Austin, T.: DIVA: A dynamic approach to microprocessor verification. J Instruction-Level Parallelism 2, 2000
8. Beatty, D., Bryant, R.: Formally verifying a microprocessor using a simulation methodology. In: ACM/IEEE Design Automation Conference, pp. 596–602, 1994

9. Berezin, S., Biere, A., Clarke, E., Zhu, Y.: Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In: Gopalakrishnan, G., Windley, P., (eds.), Formal Methods in Computer-Aided Design (FMCAD), Lecture Notes in Computer Science, vol. 1522. Springer, Berlin Heidelberg New York, 1998, pp. 369–386
10. Bose, S., Fisher, A.: Verifying pipelined hardware using symbolic logic simulation. In: International Conference on Computer Design, pp. 217–221, 1989
11. Bryant, R., German, S., Velev, M.: Processor verification using efficient decision procedures for a logic of uninterpreted functions. In: Murray, N.V., (ed.), Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX '99), Lecture Notes in Computer Science, vol. 1617. Springer, Berlin Heidelberg New York, 1999, pp. 1–13
12. Burch, J.: Techniques for verifying superscalar microprocessors. In: ACM/IEEE Design Automation Conference, pp. 552–557, 1996
13. Burch, J., Dill, D.: Automatic verification of pipelined microprocessor control. In: Dill, D.L., (ed.), Workshop on Computer-Aided Verification (CAV), Lecture Notes in Computer Science, vol. 818. Springer, Berlin Heidelberg New York, 1994, pp. 68–80
14. Damm, W., Pnueli, A.: Verifying out-of-order executions. In: Li, H.F., Probst, D.K., (eds.), IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME), pp. 23–47. Chapman and Hall, London, 1997
15. Day, N.A., Aagaard, M.D., Lou, M.: A formal analysis of the will-retire correctness statement. Technical Report 2002-14, University of Waterloo, Department of Computer Science, 2002
16. Day, N.A., Aagaard, M.D., Lou, M.: A mechanized theory for microprocessor correctness statements. Technical Report 2002-11, University of Waterloo, Department of Computer Science, 2002
17. Fox, A., Harman, N.: Algebraic models of correctness for microprocessors. Formal Aspects of Computing 12(4):298–312, 2000
18. Henzinger, T., Qadeer, S., Rajamani, S.: You assume, we guarantee: methodology and case studies. In: Hu, A.J., Vardi, M.Y., (eds.), Workshop on Computer-Aided Verification (CAV), Lecture Notes in Computer Science, vol. 1427. Springer, Berlin Heidelberg New York, 1998, pp. 440–451
19. Hosabettu, R., Gopalakrishnan, G., Srivas, M.: Verifying advanced microarchitectures that support speculation and exceptions. In: Emerson, E.A., Sistla, A.P., (eds.), Workshop on Computer-Aided Verification (CAV), Lecture Notes in Computer Science, vol. 1855. Springer, Berlin Heidelberg New York, 2000, pp. 521–537
20. Hosabettu, R., Srivas, M., Gopalakrishnan, G.: Decomposing the proof of correctness of pipelined microprocessors. In: Hu, A.J., Vardi, M.Y., (eds.), Workshop on Computer-Aided Verification (CAV), Lecture Notes in Computer Science, vol. 1427. Springer, Berlin Heidelberg New York, 1998, pp. 122–134
21. Hosabettu, R., Srivas, M., Gopalakrishnan, G.: Proof of correctness of a processor with reorder buffer using the completion functions approach. In: Halbwachs, N., Peled, D., (eds.), Workshop on Computer-Aided Verification (CAV), Lecture Notes in Computer Science, vol. 1633. Springer, Berlin Heidelberg New York, 1999, pp. 47–59
22. Jones, R., Skakkebæk, J., Dill, D.: Reducing manual abstraction in formal verification of out-of-order execution. In: Gopalakrishnan, G., Windley, P., (eds.), Formal Methods in Computer-Aided Design (FMCAD), Lecture Notes in Computer Science, vol. 1522. Springer, Berlin Heidelberg New York, 1998, pp. 2–17
23. Lahiri, S., Pixley, C., Albin, K.: Experience with term-level modeling and verification of the M*CORE microprocessor core. In: 6th Annual IEEE International Workshop on High Level Design, Validation and Test (HLDVT'01), pp. 109–114, 2001
24. Manolios, P.: Correctness of pipelined machines. In: Hunt, W.A., Johnson, S.D., (eds.), Formal Methods in Computer-Aided Design (FMCAD), Lecture Notes in Computer Science, vol. 1954. Springer, Berlin Heidelberg New York, 2000, pp. 161–178
25. McMillan, K.: Verification of an implementation of Tomasulo's algorithm by compositional model checking. In: Hu, A.J., Vardi, M.Y., (eds.), Workshop on Computer-Aided Verification (CAV), Lecture Notes in Computer Science, vol. 1427. Springer, Berlin Heidelberg New York, 1998, pp. 110–121
26. Milner, R.: An algebraic definition of simulation between programs. In: Joint Conference on Artificial Intelligence, pp. 481–489. British Computer Society, London, 1971
27. Mneimneh, M., Aloul, F., Weaver, C., Chatterjee, S., Sakallah, K., Austin, T.: Scalable hybrid verification of complex microprocessors. In: ACM/IEEE Design Automation Conference, pp. 41–46, 2001
28. Nalumasu, R., Gopalakrishnan, G.: Deriving efficient cache coherence protocols through refinement. In: Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'98), Lecture Notes in Computer Science, vol. 1388. Springer, Berlin Heidelberg New York, 1998, pp. 857–870
29. Park, S., Dill, D.: Protocol verification by aggregation of distributed transactions. In: Alur, R., Henzinger, T.A., (eds.), Workshop on Computer-Aided Verification (CAV), Lecture Notes in Computer Science, vol. 1102. Springer, Berlin Heidelberg New York, 1996, pp. 300–310
30. Patankar, V., Jain, A., Bryant, R.E.: Formal verification of an ARM processor. In: International Conference on VLSI Design, pp. 282–287. IEEE, New York, 1999
31. Pnueli, A., Arons, T.: Verification of data-insensitive circuits: An in-order-retirement case study. In: Gopalakrishnan, G., Windley, P., (eds.), Formal Methods in Computer-Aided Design (FMCAD), Lecture Notes in Computer Science, vol. 1522. Springer, Berlin Heidelberg New York, 1998, pp. 351–368
32. Qadeer, S.: Algorithms and methodology for scalable model checking. PhD thesis, Elec. Eng., Comp. Sci., University of California at Berkeley, 1999
33. Sawada, J., Hunt, W.: Trace-table-based approach for pipelined microprocessor verification. In: Grumberg, O., (ed.), Workshop on Computer-Aided Verification (CAV), Lecture Notes in Computer Science, vol. 1254. Springer, Berlin Heidelberg New York, 1997, pp. 364–375
34. Sawada, J., Hunt, W.: Processor verification with precise exceptions and speculative execution. In: Hu, A.J., Vardi, M.Y., (eds.), Workshop on Computer-Aided Verification (CAV), Lecture Notes in Computer Science, vol. 1427. Springer, Berlin Heidelberg New York, 1998, pp. 135–146
35. Sawada, J., Hunt, W.: Results of the verification of a complex pipelined machine model. In: Pierre, L., Kropf, T., (eds.), IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME), Lecture Notes in Computer Science ,vol. 1703. Springer, Berlin Heidelberg New York, 1999, pp. 313–316
36. Shen, X., Arvind: A methodology for designing correct cache coherence protocols for DSM systems. Technical Report CSG Memo 398 (A), MIT, Mass., 1997
37. Skakkebæk, J., Jones, R., Dill, D.: Formal verification of out-of-order execution using incremental flushing. In: Hu, A.J., Vardi, M.Y., (eds.), Workshop on Computer-Aided Verification (CAV), Lecture Notes in Computer Science, vol. 1427. Springer, Berlin Heidelberg New York, 1998, pp. 98–109
38. Srivas, M., Bickford, M.: Formal verification of a pipelined microprocessor. IEEE Trans Software Eng pp. 52–64, 1990
39. Srivas, M.K., Miller, S.P.: Applying formal verification to a commercial microprocessor. In: Computer Hardware Description Languages, pp. 493–502, 1995
40. Velev, M., Bryant, R.: Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. In: ACM/IEEE Design Automation Conference, pp. 41–46, 2000
41. Windley, P., Coe, M.: A correctness model for pipelined microprocessors. In: Theorem Provers in Circuit Design, pp. 32–51. Springer, Berlin Heidelberg New York, 1994