

# Relating Multi-step and Single-Step Microprocessor Correctness Statements

Mark D. Aagaard<sup>1</sup>, Nancy A. Day<sup>2</sup>, and Meng Lou<sup>2</sup>

<sup>1</sup> Electrical and Computer Engr., University of Waterloo  
markaa@swen.uwaterloo.ca

<sup>2</sup> Computer Science, University of Waterloo, Waterloo, ON, Canada  
nday@cs.uwaterloo.ca, mlou@student.math.uwaterloo.ca

**Abstract.** A diverse collection of correctness statements have been proposed and used in microprocessor verification efforts. Correctness statements have evolved from criteria that match a single step of the implementation against the specification to seemingly looser, multi-step, criteria. In this paper, we formally verify conditions under which two categories of multi-step correctness statements logically imply single-step correctness statements. The first category of correctness statements compare flushed states of the implementation and the second category compare states that are able to retire instructions. Our results are applicable to superscalar implementations, which fetch or retire multiple instructions in a single step.

## 1 Introduction

Microprocessor verification efforts usually compare a state-machine description of a microarchitectural-level implementation against an Instruction Set Architecture (ISA). The correctness statement describes the intended relationship between the implementation and the specification ISA. In early verification efforts, correctness statements were based on Milner's pointwise notion of simulation — a commuting diagram that says for any step the implementation takes, the specification must take a corresponding step [15]. Pipelining and other optimizations increased the gap between the behaviour of the implementation and the specification, making it more difficult to show that an individual implementation step corresponds to a specification step. In a seminal paper, Burch and Dill proposed constructing abstraction functions automatically by flushing pipelines [5]. Their correctness criteria compares each step of the implementation against the specification by flushing the implementation. As verification efforts have tackled complexities such as out-of-order execution and interrupts, the correctness statements have evolved from single-step criteria to seemingly looser, multi-step criteria. Sawada and Hunt [16], Hosabetu *et al.* [10], Jones *et al.* [14], and Arons and Pnueli [3] check that the implementation corresponds with the specification only at flushed implementation states, i.e. states with no in-flight instructions. Fox and Harman [7] compare the implementation and specification only at states where an instruction is about to retire. Berezin *et al.* [4] compare multi-step implementation traces that fetch a single instruction against a single step of the specification.

The change from single-step to multi-step correctness statements raises the questions “are they proving the same relationship?”, “are there correct machines that satisfy multi-step correctness but not single-step?”, and finally, “are there bugs that are undetectable with multi-step correctness statements?” To explore the relationship between multi-step and single-step correctness statements, we build on the Microbox framework [1,2] for microprocessor correctness statements. Using Microbox, Aagaard *et al.* [2] described and compared thirty-seven correctness statements from twenty-nine papers. Day *et al.* [6] mechanized Microbox in the HOL theorem prover [8] and verified a partial order relationship between correctness statements. Day *et al.* proved that tighter criteria, such as single-step correctness statements, logically imply looser criteria, such as testing only flushed states of the implementation. In this paper we examine whether some reverse implications hold, i.e., if a multi-step correctness statement is verified, is there a single-step statement that also holds?

Section 2 provides background material on Microbox. Section 3 characterizes the microprocessor-specific functions used in the correctness statements. Section 4 describes the relationship between multi-step correctness that compares flushed states and single-step correctness using Burch-Dill style flushing. The main result of the section is Theorem 3, which says that comparing flushed states of the implementation against the specification is equivalent to using flushing to compare each step of the implementation, for deterministic specifications with no internal state. We also provide an example of a non-deterministic specification and implementation that satisfy the multi-step correctness statement, but not the single-step statement with flushing. Section 5 describes the relationship between multi-step correctness at retirement to single-step correctness. Theorem 6 says that comparing the implementation to the specification when instructions are about to retire is equivalent to checking each step of the implementation. Our results are applicable to superscalar implementations, which can fetch and retire multiple instructions in a single step. Sections 6 and 7 consider the relevance of our results to existing verification efforts and summarize the paper.

## 2 The Microbox Framework

The Microbox framework uses four parameters to characterize a correctness statement: alignment, match, implementation execution, and specification execution. *Alignment* is the method used to align the traces of the implementation and specification (Section 2.1). *Match* is the relation established between aligned states in the implementation and specification traces (Section 2.2). *Implementation execution* and *specification execution* describe the type of state machines used – either deterministic or non-deterministic. The Microbox framework provides a list of options for each of these parameters based on verification efforts discussed in the literature (Table 1). By choosing options for the parameters, Microbox can produce a wide variety of correctness statements.

Each correctness statement contains a base case and an induction step. The base cases deal with initial states and are generally quite straightforward, so we concentrate on the induction steps. The alignment parameter determines the overall form of the induction clause. For each alignment option, Microbox defines a correctness statement for an other match (O), non-deterministic implementation (N), and non-deterministic specification

**Table 1.** Options for correctness statement parameters

$\langle alignment \rangle$	$\langle match \rangle$	$\langle impl. execution \rangle$	$\langle spec. execution \rangle$
(F) Flushpoint	(O) Other	(N) Non-deterministic	(N) Non-deterministic
(W) Will-retire	(A) Abstraction	(D) Deterministic	(D) Deterministic
(M) Must-issue	(U) Flushing		
(S) Stuttering	(E) Equality		
(I) Informed-pointwise	(R) Refinement Map		
(P) Pointwise			

Example: IUND = informed-pointwise alignment (I), flushing match (U), non-deterministic implementation (N) and deterministic specification (D).

(N). Correctness statements for different match and execution options are generated by substitutions into the \*ONN definitions.

In Microbox, both the specification and implementation machines have program memories as part of their state, and so do not take instructions as inputs. Invariants, which limit the state space of a machine to reachable states or an over-approximation of reachable states, are encoded in the set of states for a machine. Table 2 summarizes the notation.

**Table 2.** State-machine notation

$N$	Next-state relation
$N^k(q, q')$	$q'$ is reachable from $q$ in $k$ steps of $N$
$n$	Next-state function
$\pi$	External state projection function.
$q_i \stackrel{\pi}{=} q_s$	Externally visible equivalence: $\pi_i(q_i) = \pi_s(q_s)$ .

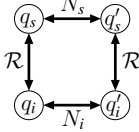
Identifiers are subscripted with “s” for specification and “i” for implementation.

In Sections 2.1 and 2.2, we describe the alignment and match options that are relevant to this paper. In Section 2.3, we characterize the correctness statements in terms of the type of synchronization used, i.e. at fetch or at retire. In Section 2.4, we describe the partial order relationships between these correctness statements.

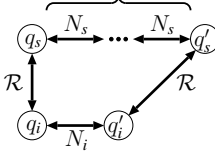
## 2.1 Alignment

Alignment describes which states in the execution trace are tested for matching. *Pointwise alignment* (P, Definition 1) is the classic commuting diagram. *Informed-pointwise* (I, Definition 2) is a variation of pointwise alignment suitable for superscalar implementations, which allows the implementation to inform the correctness statement of the number of specification steps to take. In practice, *numInstr* is instantiated with either the number of instructions that were fetched (*numFetch*) or the number of instructions that were retired (*numRetire*), depending on the synchronization method (Section 2.3).

**Definition 1 (Pointwise induction clause: PONN).**

$$\text{PONN}(\mathcal{R}, N_i, N_s) \equiv \forall q_i, q'_i. \forall q_s. \exists q'_s. \left[ \wedge N_i(q_i, q'_i) \wedge \mathcal{R}(q_i, q_s) \right] \implies \left[ \wedge N_s(q_s, q'_s) \wedge \mathcal{R}(q'_i, q'_s) \right]$$


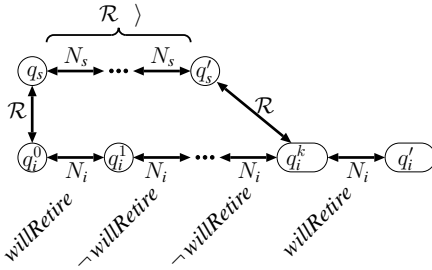
**Definition 2 (Informed-pointwise induction clause: IONN).**

$$\text{IONN}(\text{numInstr}, \mathcal{R}, N_i, N_s) \equiv \forall q_i, q'_i. \forall q_s. \exists q'_s. \text{let } j = \text{numInstr}(q_i, q'_i) \text{ in } \left[ \wedge N_i(q_i, q_s) \wedge \mathcal{R}(q_i, q_s) \right] \implies \left[ \wedge N_s^j(q_s, q'_s) \wedge \mathcal{R}(q'_i, q'_s) \right]$$


*Will-retire alignment* (W, Definition 3) compares the implementation and specification whenever the implementation is ready to retire instructions. The implementation retires one or more instructions in the first step of the trace and continues until it is ready to retire again.

**Definition 3 (Will-retire induction clause: WONN).**

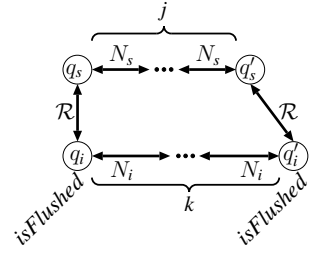
$$\text{WONN}(\text{numRetire}, \text{willRetire}, \mathcal{R}, N_i, N_s) \equiv \forall q_i^0, q_i^1, \dots, q_i^k. \forall q_s. \exists q'_s. \text{let } r = \text{numRetire}(q_i^0, q_i^1) \text{ in } \left[ \wedge N_i(q_i^0, q_i^1) \wedge \text{willRetire}(q_i^0, q_i^1) \wedge \left( \forall j \in 1 \dots k - 1. N_i(q_i^j, q_i^{j+1}) \wedge \neg \text{willRetire}(q_i^j, q_i^{j+1}) \right) \wedge (\exists q'_i. N_i(q_i^k, q'_i) \wedge \text{willRetire}(q_i^k, q'_i)) \wedge \mathcal{R}(q_i^0, q_s) \right] \implies \left[ \wedge N_s^r(q_s, q'_s) \wedge \mathcal{R}(q'_i, q'_s) \right]$$



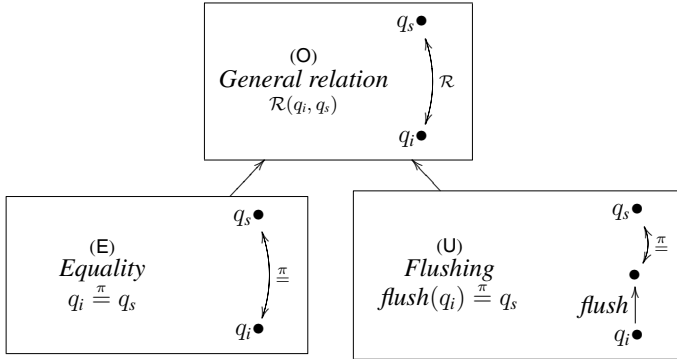
*Flushpoint alignment* (F, Definition 4) compares flushed states of the implementation against the specification. It says that if there is a trace between flushed implementation states, then there must exist a trace in the specification between a pair of states that match the flushed implementation states.

**Definition 4 (Flushpoint induction clause: FONN).**

$$\text{FONN}(isFlushed, \mathcal{R}, N_i, N_s) \equiv \forall q_i, q'_i, q_s. \exists q'_s. \left[ \begin{array}{l} isFlushed(q_i) \\ \wedge \exists k. N_i^k(q_i, q'_i) \\ \wedge isFlushed(q'_i) \\ \wedge \mathcal{R}(q_i, q_s) \end{array} \right] \implies \left[ \wedge \exists j. N_s^j(q_s, q'_s) \right]$$


**2.2 Match**

Instantiations for the match parameter are relations between an implementation state  $q_i$  and specification state  $q_s$  that mean “ $q_i$  is a correct representation of  $q_s$ ”. Figure 1 shows the match options that are relevant to this paper and the partial order on the options.



**Fig. 1.** Options and partial order for the match parameter

An *other match* (O) is any relation between implementation and specification states. The *flushing match* (U) uses a flushing function to compute an implementation state that should be externally equivalent to a specification state. The *equality match* (E) requires that the implementation and specification states be externally equivalent.

**2.3 Synchronization**

In the implementation projection function ( $\pi_i$ ), there are two common representations of the program counter: the address of the next instruction to fetch, and the address of the next instruction to retire. We refer to the first option as *synchronization at fetch* and the second option as *synchronization at retirement*.

For a projection function to be sensible, the program counter, register file, and other state components must all reflect the same point in the execution of a program. Synchronization at fetch is only appropriate when applied to a flushed implementation state.

Hence, synchronization at fetch can only be used with the flushing match, which flushes the implementation before applying the projection function, and with flushpoint alignment. With synchronization at retirement, the register file and program counter always correspond to the same point of execution.

The function *numInstr* is instantiated with *numFetch* for synchronization at fetch and *numRetire* for synchronization at retirement. Instructions in the shadow of a mis-predicted branch or an exception should not be executed by the specification, and so do not count toward the number of instructions fetched. The function *numRetire* counts the number of instructions that retire. Every instruction that retires should be executed by the specification.

## 2.4 Correctness Space

Figure 2 shows the partial order of logical implication for the first two parameters of correctness statements (alignment and match). For the third and fourth parameters, the execution of the implementation and specification machines, it is easy to consider deterministic as an instance of non-deterministic, thereby providing the ordering amongst these options. The alignment parameter *iF* (Definition 5, informed-flushpoint — a common instance of *F*) will be introduced in Section 4.1. The non-shaded lines show the natural ordering amongst correctness criteria, which was verified in Day *et al.* [6].

In this paper, we verify the arrows in the shaded boxes, which proves equivalences between the correctness statements. In Section 4.2, we verify informed-flushpoint with the equality match for deterministic specifications with no internal state is equivalent to informed-pointwise with flushing ( $iFE \iff IU$ ). The dashed line between *iFE* and *IU* indicates that this implication holds only for deterministic specifications. In Section 5, we prove will-retire equality is equivalent to informed-pointwise equality ( $WE \iff IE$ ).

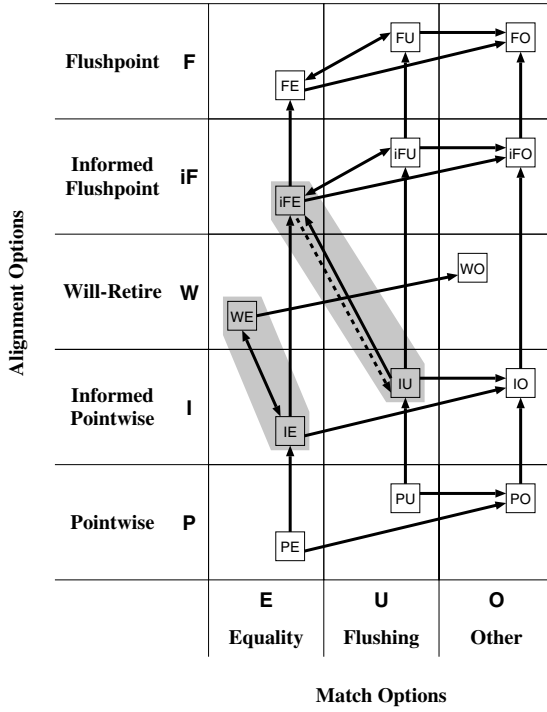
In related work, we verified that the multi-step correctness statement of must-issue with the flushing match, in which the implementation takes some number of stalled steps followed by one step where it fetches an instruction, is equivalent to the single-step informed-pointwise flushing (*IUNN*) [6].

## 3 Characterization of Microprocessor-Specific Functions

The relationships between correctness statements are based on microprocessor-specific functions and relations (Table 3) behaving appropriately. In this section, we describe the required conditions on these functions. These conditions often appear as lemmas in verification efforts. To apply our results to a particular specification and implementation, these conditions would have to be verified. Conditions 1–5 are for synchronization at fetch. Conditions 6–8 are for synchronization at retirement.

### 3.1 Fetching and Flushing Conditions

Condition 1 states that *numFetch* is zero in a step if-and-only-if *doesFetch* is false.


**Fig. 2.** Partial order for correctness statements

**Table 3.** Microprocessor-specific functions

$doesFetch(q_i, q'_i)$	true if an instruction is fetched in a step.
$numFetch(q_i, q'_i)$	returns the number of instructions fetched in a step.
$willRetire(q_i, q'_i)$	true if an instruction is retired in a step.
$numRetire(q_i, q'_i)$	returns the number of instructions retired in a step.
$flush(q_i)$	flushes $q_i$ , i.e., completes the execution of any in-flight instructions.
$isFlushed(q_i)$	true if a state is flushed.

**Condition 1 (numFetch and doesFetch)**

$$numFetch\_doesFetch(numFetch, doesFetch) \equiv \forall q_i, q'_i. (numFetch(q_i, q'_i) = 0) \iff \neg doesFetch(q_i, q'_i)$$

We characterize the required behaviour of a flushing function with Conditions 2 and 3. Condition 2 relates the function  $flush$  to the predicate  $isFlushed$  and says that if a state  $q_i$  is flushed, then flushing  $q_i$  returns  $q_i$ , i.e.  $flush$  is the identity function for a flushed state.

**Condition 2 (isFlushed and flush)**

$$isFlushed\_flush(isFlushed, flush) \equiv \forall q_i. isFlushed(q_i) \implies (flush(q_i) = q_i)$$

Condition 3 says that if an instruction is not fetched in a step where the implementation transitions from  $q_i$  to  $q'_i$ , then flushing  $q_i$  returns the same state as flushing  $q'_i$ . Equivalently, flushing a stalled state results in the same state as allowing the machine to take one (unproductive) step and then flushing.

**Condition 3 (doesFetch and flush)**

$$\begin{aligned} doesFetch\_flush(doesFetch, flush, N_i) \equiv \\ \forall q_i, q'_i. \neg doesFetch(q_i, q'_i) \wedge N_i(q_i, q'_i) \implies (flush(q_i) = flush(q'_i)) \end{aligned}$$

Conditions 2 and 3 are the only restrictions on flushing functions. The construction of the flushing function is up to the verifier. The most common method for constructing a flushing function was originated by Burch and Dill [5]. They iterate a deterministic implementation's next-state function without fetching new instructions. Another method for constructing flushing functions was developed by Hosabettu *et al.* [10], who define completion functions for each stage in the pipeline and then compose the completion functions to create a flushing function.

We also need a reachability condition and a liveness condition. Condition 4 says that for any implementation state,  $q_i$ , there exists a trace from a flushed implementation state to  $q_i$ .

**Condition 4 (Past Flush)**

$$\begin{aligned} past\_flush(isFlushed, N_i) \equiv \\ \forall q_i. \exists k, q_i^0. isFlushed(q_i^0) \wedge N_i^k(q_i^0, q_i) \end{aligned}$$

Condition 5 says that from any state, the implementation can reach a flushed state by passing through a series of states where it does not fetch an instruction. If the implementation does not already have the ability to prevent instructions from being fetched, then flushing circuitry must be added.

**Condition 5 (Eventually Flushed)**

$$\begin{aligned} eventually\_flushed(isFlushed, doesFetch, N_i) \equiv \\ \forall q_i. \exists k, q_i^0, \dots, q_i^k. \\ q_i = q_i^0 \wedge (\forall j < k. N_i(q_i^j, q_i^{j+1}) \wedge \neg doesFetch(q_i^j, q_i^{j+1})) \wedge isFlushed(q_i^k) \end{aligned}$$

**3.2 Retiring and Projection Conditions**

Condition 6 states that  $numRetire$  is zero for an implementation step if-and-only-if  $willRetire$  is false. It is the dual of Condition 1 for synchronization at retirement.

**Condition 6 (numRetire and willRetire)**

$$\begin{aligned} numRetire\_willRetire(numRetire, willRetire) \equiv \\ \forall q_i, q'_i. (numRetire(q_i, q'_i) = 0) \iff \neg willRetire(q_i, q'_i) \end{aligned}$$



Condition 7, relating the predicate *willRetire* to the implementation projection function  $\pi_i$  appropriate for synchronization at retirement, is the dual of Condition 3. Condition 7 says that if an instruction is not retired in a step where the implementation transitions from  $q_i$  to  $q'_i$ , then the projections of  $q_i$  and  $q'_i$  are equivalent.

**Condition 7 (*willRetire* and  $\pi_i$ )**

$$\begin{aligned} & \text{willRetire\_pi}(\text{willRetire}, \pi_i, N_i) \equiv \\ & \forall q_i, q'_i. \neg \text{willRetire}(q_i, q'_i) \wedge N_i(q_i, q'_i) \implies (\pi_i(q_i) = \pi_i(q'_i)) \end{aligned}$$

Condition 8 is a liveness condition. The condition says that from any implementation state, it is possible to reach a state that can retire an instruction.

**Condition 8 (Eventually Retires)**

$$\begin{aligned} & \text{eventually\_retires}(\text{willRetire}, N_i) \equiv \\ & \forall q_i. \exists k, q_i^*, q'_i. N_i^k(q_i, q_i^*) \wedge N_i(q_i^*, q'_i) \wedge \text{willRetire}(q_i^*, q'_i) \end{aligned}$$

## 4 Flushpoint Equality and Informed-Pointwise Flushing

In this section, we discuss the relationship between the two correctness statements, flushpoint equality (FE) and informed-pointwise flushing (IU), which use synchronization at fetch. IU is Burch-Dill style flushing. In Section 4.1, we introduce a commonly used version of flushpoint alignment, which we call informed-flushpoint (iF). In Section 4.2, we prove that informed-flushpoint equality and informed-pointwise flushing are equivalent for a *deterministic* specification with no internal state ( $\text{iFEND} \iff \text{IUND}$ , Theorem 3). A similar relationship does not exist between flushpoint equality (FE) and informed-pointwise flushing (IU), because flushpoint alignment does not constrain the number of steps in the specification trace. In Section 4.3, we describe an implementation and a *non-deterministic* specification that satisfy informed-flushpoint equality but not informed-pointwise flushing, thereby providing a counterexample to  $\text{iFENN} \implies \text{IUNN}$ .

### 4.1 Informed-Flushpoint

Flushpoint alignment (Definition 4) does not impose any constraints on the number of specification steps taken. However, in most verification efforts that use flushpoint alignment (e.g., [16, 10, 14]), the number of steps in the specification trace is the number of instructions executed in the implementation trace. We introduce *informed-flushpoint* alignment (iF) to capture this common practice. Informed-flushpoint is most commonly used with the equality match, as shown in Definition 5. We overload *numFetch* to return the total number of instructions fetched in either a sequence of implementation steps or in a single implementation step.

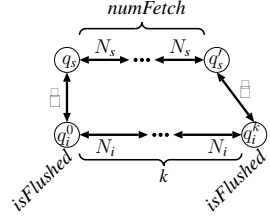
**Definition 5 (Informed-Flushpoint Equality induction clause: iFENN).**

$$\text{iFENN}(isFlushed, numFetch, \pi_i, \pi_s, N_i, N_s) \equiv$$

$$\forall q_i^0, q_i^1, \dots, q_i^k. \forall q_s. \exists q'_s.$$

$$\text{let } f = numFetch\langle q_i^0, \dots, q_i^k \rangle \text{ in}$$

$$\left[ \begin{array}{l} isFlushed(q_i^0) \\ \wedge (\forall j < k. N_i(q_i^j, q_i^{j+1})) \\ isFlushed(q_i^k) \\ \wedge q_i^0 \stackrel{\pi}{=} q_s \end{array} \right] \Longrightarrow \left[ \begin{array}{l} N_s^f(q_s, q'_s) \\ q_i^k \stackrel{\pi}{=} q'_s \end{array} \right]$$

**4.2 Informed-Flushpoint and Informed Pointwise: Deterministic Specification**

In this section, we prove Theorem 3, which says that, for a deterministic specification without internal state (i.e.  $N_s$  is  $n_s$  and  $\pi_s$  is identity), informed-flushpoint with the equality match (iFEND, an instantiation of Definition 5) is equivalent to informed-pointwise with the flushing match (IUND, an instantiation of Definition 2). Showing that the single-step informed-pointwise correctness statement logically implies multi-step informed-flushpoint (IUND  $\implies$  iFEND) is straightforward by induction. Here we describe the more difficult reverse direction (iFEND  $\implies$  IUND). First, we introduce an intermediate point, which we call iFflush (Definition 6) and prove iFEND  $\implies$  iFflush (Theorem 1). Second, we show iFflush  $\implies$  IUND (Theorem 2).

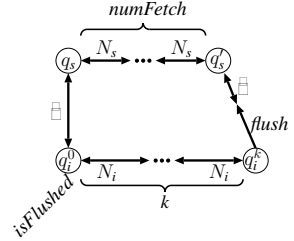
**Definition 6 (iFflush).**

$$\text{iFflush}(isFlushed, numFetch, flush, \pi_i, \pi_s, N_i, N_s) \equiv$$

$$\forall q_i^0, q_i^1, \dots, q_i^k. \forall q_s. \exists q'_s.$$

$$\text{let } f = numFetch\langle q_i^0, \dots, q_i^k \rangle \text{ in}$$

$$\left[ \begin{array}{l} isFlushed(q_i^0) \\ \wedge (\forall j < k. N_i(q_i^j, q_i^{j+1})) \\ \wedge q_i \stackrel{\pi}{=} q_s \end{array} \right] \Longrightarrow \left[ \begin{array}{l} N_s^f(q_s, q'_s) \\ flush(q_i^k) \stackrel{\pi}{=} q'_s \end{array} \right]$$



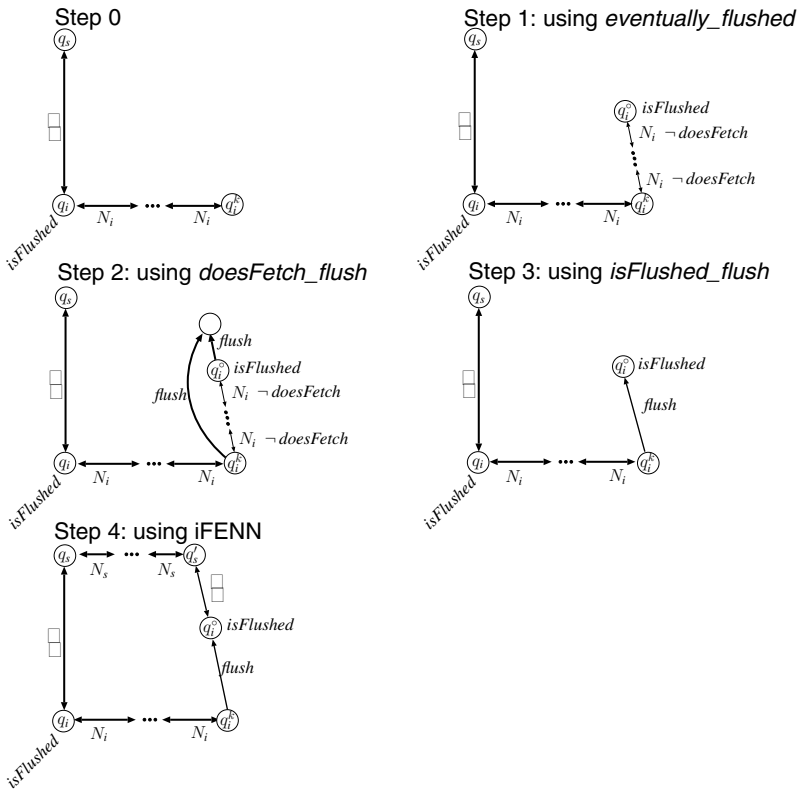
Definition 6 is the same as informed-flushpoint (Definition 5), except that the final states must satisfy the flushing match, rather than be externally equivalent.

**Theorem 1 (iFENN  $\implies$  iFflush).**

$$\forall isFlushed, numFetch, doesFetch, flush, \pi_i, \pi_s, N_i, N_s.$$

$$\left[ \begin{array}{ll} \text{eventually\_flushed}(isFlushed, doesFetch, N_i) & \text{--- Condition 5} \\ \wedge \text{doesFetch\_flush}(doesFetch, flush, N_i) & \text{--- Condition 3} \\ \wedge \text{isFlushed\_flush}(isFlushed, flush) & \text{--- Condition 2} \\ \wedge \text{numFetch\_doesFetch}(numFetch, doesFetch) & \text{--- Condition 1} \end{array} \right] \Longrightarrow \left[ \begin{array}{l} \text{iFENN}(isFlushed, numFetch, \pi_i, \pi_s, N_i, N_s) \\ \implies \text{iFflush}(isFlushed, numFetch, flush, \pi_i, \pi_s, N_i, N_s) \end{array} \right]$$

Figure 3 outlines the proof of  $iFENN \implies iFlush$  (Theorem 1). This theorem depends on conditions described in Section 3. We begin in Step 0 assuming the left and lower sides of the commuting diagram for  $iFlush$ . In Step 1, we extend the path from  $q_i^k$  to a flushed state,  $q_i^o$ , using the condition that the implementation can always reach a flushed state by taking steps that do not fetch instructions (Condition 5, *eventually\_flushed*). In Step 2, we use the condition that flushing a state after taking a series of steps that do not fetch an instruction is the same as flushing the state at the beginning of the series (Condition 3, *doesFetch\_flush*). In Step 3, we conclude that flushing  $q_i^k$  results in  $q_i^o$  because flushing a flushed state has no effect (Condition 2, *isFlushed\_flush*). In Step 4, we use the fact that  $iFENN$  holds for traces between flushed states to complete the commuting diagram. Condition 1, which relates *numFetch* and *doesFetch*, is needed to relate the number of steps in the specification traces.



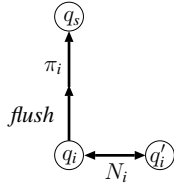
**Fig. 3.** Steps in proof of  $iFENN \implies iFlush$  (Theorem 1)

In the second half of the proof of  $iFEND \implies IUND$ , we use  $iFlush$  to arrive at  $IUND$  (Theorem 2). The steps of the proof are outlined in Figure 4.

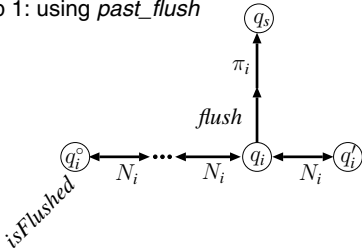
**Theorem 2 (iFlush  $\implies$  IUND).**

$$\begin{aligned} & \forall \text{ isFlushed}, \text{ numFetch}, \text{ flush}, \pi_i, \pi_s, N_i, n_s. \\ & \left[ \begin{aligned} & \wedge \text{ past\_flush}(\text{isFlushed}, N_i) \\ & \wedge \pi_s = (\lambda x.x) \end{aligned} \right. \quad \text{--- Condition 4} \left. \right] \\ & \implies \left[ \begin{aligned} & \text{iFlush}(\text{isFlushed}, \text{ numFetch}, \text{ flush}, \pi_i, \pi_s, N_i, n_s) \\ & \implies \text{IUND}(\text{ numFetch}, \text{ flush}, \pi_i, \pi_s, N_i, n_s) \end{aligned} \right] \end{aligned}$$

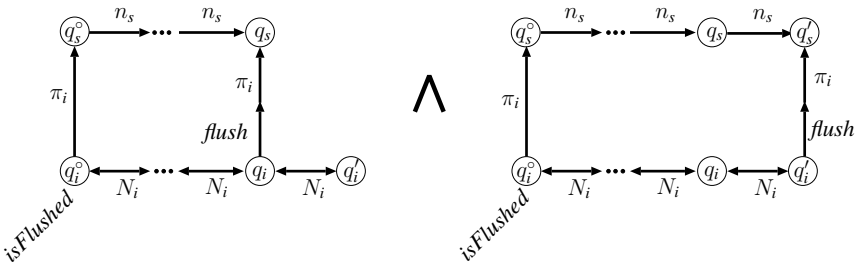
Step 0



Step 1: using past\_flush



Step 2: using iFlush twice



Step 3: IUND

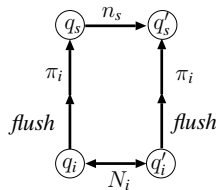


Fig. 4. Steps in proof of iFlush  $\implies$  IUND (Theorem 2)

In Step 0 of Figure 4, we start with the left and lower edges of the IUND commuting diagram, leaving out  $\pi_s$  because it is the identity function in this case. In Step 1, we extend the path from  $q_i$  back to a flushed state,  $q_i^\circ$ , using the condition that for any state, there is always a previous flushed state (Condition 4, *past.flush*). In Step 2, we use *iflush* to deduce the two commuting diagrams both beginning at  $q_i^\circ$ . Because the matching relationship is a function, and because the specification is deterministic, from these two commuting diagrams we can conclude IUND in Step 3.

We combine Theorem 1, specialized for a deterministic specification with no internal state; Theorem 2; and the result that IUND logically implies *ifEND* to conclude that *ifEND* is equivalent to IUND under the conditions listed in Section 3 (Theorem 3).

**Theorem 3 (*ifEND*  $\iff$  IUND).**

$$\forall \text{ isFlushed, numFetch, doesFetch, flush, } \pi_i, \pi_s, N_i, n_s.$$

$\wedge$	<i>eventually.flushed</i> ( <i>isFlushed</i> , <i>doesFetch</i> , $N_i$ )	— Condition 5
$\wedge$	<i>doesFetch.flush</i> ( <i>doesFetch</i> , <i>flush</i> , $N_i$ )	— Condition 3
$\wedge$	<i>isFlushed.flush</i> ( <i>isFlushed</i> , <i>flush</i> )	— Condition 2
$\wedge$	<i>numFetch.doesFetch</i> ( <i>numFetch</i> , <i>doesFetch</i> )	— Condition 1
$\wedge$	<i>past.flush</i> ( <i>isFlushed</i> , $N_i$ )	— Condition 4
$\wedge$	$\pi_s = (\lambda x.x)$	

$$\implies \left[ \begin{array}{c} \text{iFEND}(\text{isFlushed}, \text{numFetch}, \pi_i, \pi_s, N_i, n_s) \\ \iff \\ \text{IUND}(\text{numFetch}, \text{flush}, \pi_i, \pi_s, N_i, n_s) \end{array} \right]$$

### 4.3 Informed-Flushpoint and Informed-Pointwise: Non-Deterministic Specification Counterexample

In Section 4.2, we proved *ifEND*  $\iff$  IUND. In this section, we illustrate that a non-deterministic specification paired with an implementation can satisfy *ifENN* without satisfying IUNN. Figure 5 is an example of a reasonable non-deterministic specification and a slightly strange, but arguably correct, implementation that satisfies *ifENN* but not IUNN. In the specification states (S1—S9), the letters in the top of the box represent instructions to execute. The lower part of the box lists completed instructions. In the implementation states (I1—I7), the middle shaded area is in-flight instructions. States with no in-flight instructions are flushed. The larger, shaded arrows show the projection of the implementation states. In the step marked “X” the implementation kills its currently executing instruction “B” and fetches the instructions “C”, and “D”, however it only reports fetching one instruction.

Figure 6 shows how the *ifENN* commuting diagram is satisfied for all possible paths between flushed implementation states. In all three cases, the length of the specification trace is the reported number of instructions fetched. Because there is a bug in the fetch mechanism, this is not actually the number of instructions fetched in Path 3. Figure 7 illustrates that IUNN does not hold for the implementation step “X”.

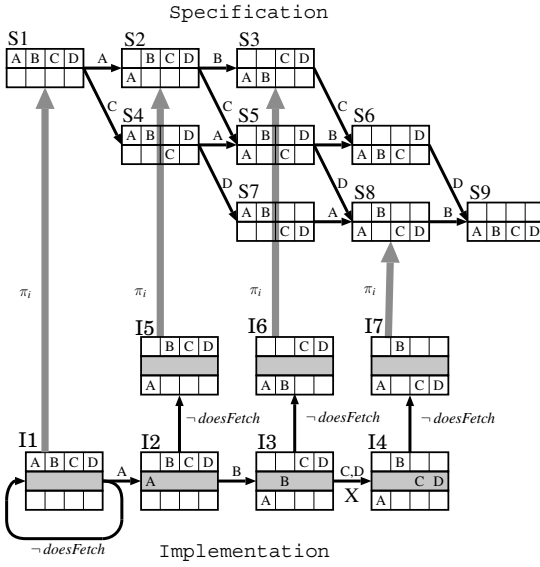


Fig. 5. Specification and implementation of counterexample

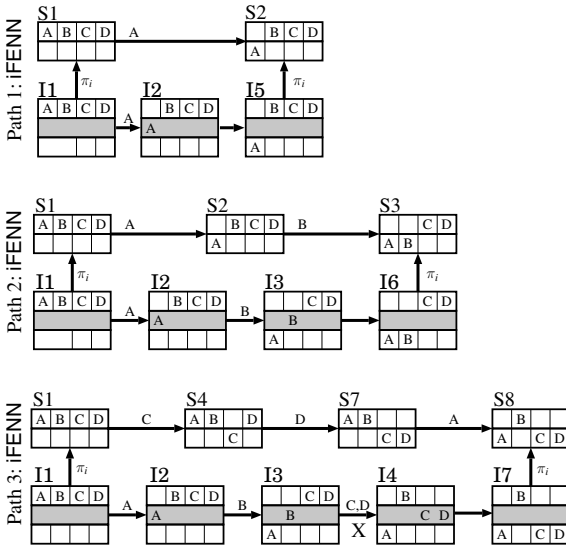


Fig. 6. iFENN paths of counterexample

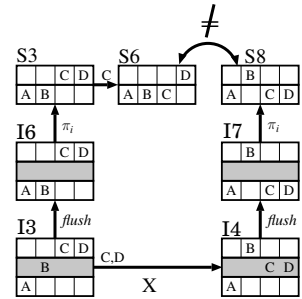


Fig. 7. IUNN path of counterexample

## 5 Will-Retire and Informed-Pointwise

The will-*retire* correctness statement (WONN, Definition 3) uses synchronization at retirement to compare an implementation trace that retires instructions only in the first step against one specification step. The implementation trace continues until it is ready to retire another instruction. The main result of this section is Theorem 6, which says that will-*retire* equality (WENN) is equivalent to informed-pointwise with equality (IENN, Definition 2 with the equality match).

The first insight in the proof that WENN is equivalent to IENN is the introduction of an alternative way of expressing WONN, which we call *single-step will-*retire** (ssWONN, Definition 7). ssWONN decomposes WONN into two simpler, single-step properties based on whether the implementation will retire any instructions.

As a single-step correctness statement, ssWONN is similar to informed-pointwise (IONN) in examining only a single step of the implementation. IONN and ssWONN are equivalent under Condition 6, *numRetire\_willRetire*, which states that the function *numRetire* returns zero if-and-only-if *willRetire* is false (Theorem 4).

### Definition 7 (Single-step will-*retire* induction clause: ssWONN).

$\text{ssWONN}(\text{numRetire}, \text{willRetire}, \mathcal{R}, N_i, N_s) \equiv$

$\forall q_i, q'_i. \forall q_s.$

let  $r = \text{numRetire}(q_i, q'_i)$  in

$$\left[ \bigwedge N_i(q_i, q'_i) \right] \implies \left[ \bigwedge \begin{array}{l} \text{willRetire}(q_i, q'_i) \implies \exists q'_s. N_s^r(q_s, q'_s) \wedge \mathcal{R}(q'_i, q'_s) \\ \neg \text{willRetire}(q_i, q'_i) \implies \mathcal{R}(q'_i, q_s) \end{array} \right]$$

### Theorem 4 (ssWONN $\iff$ IONN).

$\forall \text{numRetire}, \text{willRetire}, \mathcal{R}, N_i, N_s.$

$\text{numRetire\_willRetire}(\text{numRetire}, \text{willRetire})$  — Condition 6

$$\implies \left[ \begin{array}{c} \text{ssWONN}(\text{numRetire}, \text{willRetire}, \mathcal{R}, N_i, N_s) \\ \iff \\ \text{IONN}(\text{numRetire}, \mathcal{R}, N_i, N_s) \end{array} \right]$$

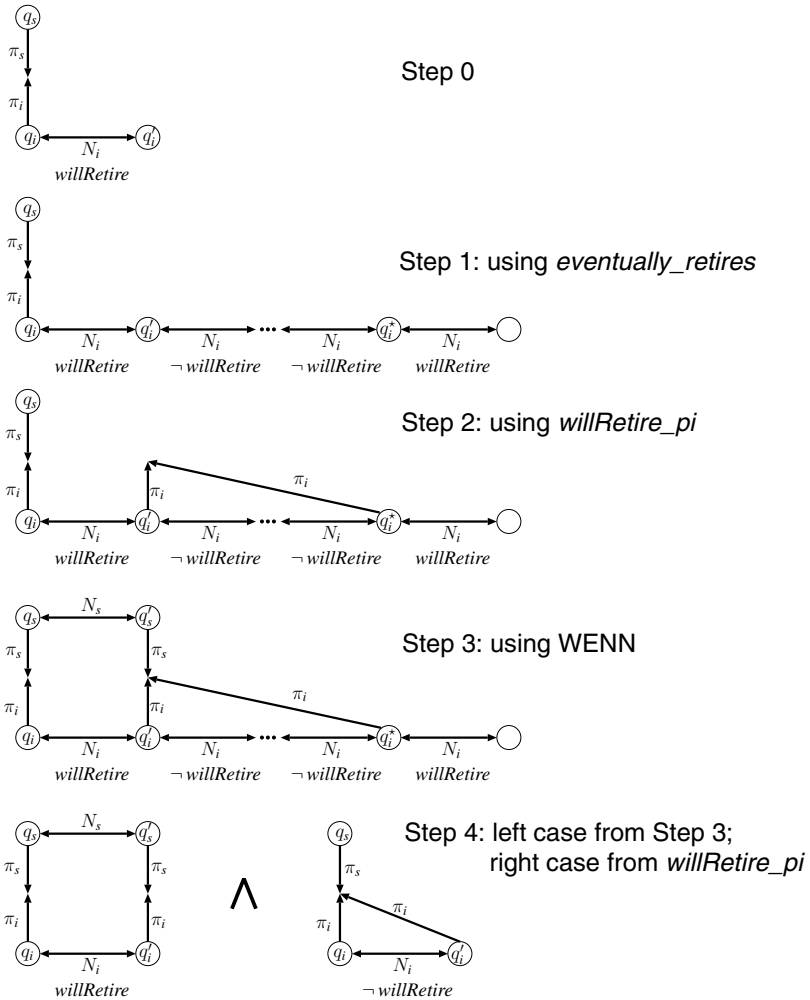
The next and more challenging step in the proof is to show that will-*retire* with the equality match is equivalent to the seemingly tighter single-step will-*retire* correctness statement (WENN  $\iff$  ssWENN). Showing  $\text{ssWENN} \implies \text{WENN}$  is straightforward by induction. The other direction ( $\text{WENN} \implies \text{ssWENN}$ , Theorem 5) holds under Conditions 7 and 8.

### Theorem 5 (WENN $\iff$ ssWENN).

$\forall \text{willRetire}, \pi_i, \pi_s, N_i, N_s.$

$$\left[ \begin{array}{l} \text{willRetire\_pi}(\text{willRetire}, \pi_i, N_i) \text{ — Condition 7} \\ \bigwedge \text{eventually\_retires}(\text{willRetire}, N_i) \text{ — Condition 8} \end{array} \right] \implies \left[ \begin{array}{c} \text{WENN}(\text{numRetire}, \text{willRetire}, \pi_i, \pi_s, N_i, N_s) \\ \iff \\ \text{ssWENN}(\text{numRetire}, \text{willRetire}, \pi_i, \pi_s, N_i, N_s) \end{array} \right]$$

Figure 8 is an illustration of the proof of Theorem 5. In Step 0, we start with the left and lower side of the commuting diagram for *ssWENN*. In Step 1, we use the eventually retires condition (Condition 8), to reach the first future state,  $q_i^*$ , that retires an instruction. In Step 2, we use the *willRetire\_pi* condition (Condition 7) to conclude the projection of  $q_i'$  and  $q_i^*$  are equal. In Step 3, we use *WENN* to complete the commuting diagram. Step 4 shows *ssWENN* where the left case follows from Step 3 and the right case follows directly from Condition 7.



**Fig. 8.** Steps in proof of *WENN*  $\implies$  *ssWENN* (Theorem 6)



**Theorem 6 (WENN  $\iff$  IENN).**

$$\begin{array}{l} \forall \text{ willRetire}, \pi_i, \pi_s, N_i, N_s. \\ \left[ \begin{array}{l} \wedge \text{ willRetire\_pi}(\text{willRetire}, \pi_i, N_i) \quad \text{--- Condition 7} \\ \wedge \text{ eventually\_retires}(\text{willRetire}, N_i) \quad \text{--- Condition 8} \\ \wedge \text{ numRetire\_willRetire}(\text{numRetire}, \text{willRetire}) \quad \text{--- Condition 6} \end{array} \right] \\ \implies \left[ \begin{array}{c} \text{WENN}(\text{numRetire}, \text{willRetire}, \pi_i, \pi_s, N_i, N_s) \\ \iff \\ \text{IENN}(\text{numRetire}, \pi_i, \pi_s, N_i, N_s) \end{array} \right] \end{array}$$

By specializing  $\mathcal{R}$  in Theorem 4 to the equality match, we are able to conclude IENN is equivalent to ssWENN under Condition 6. Combining this specialization of Theorem 4 with Theorem 5, we conclude WENN  $\iff$  IENN under Conditions 7, 8, and 6 (Theorem 6).

## 6 Relating Theory to Practice

We now consider the relevance of our results to existing microprocessor verification efforts that use multi-step correctness statements based on flushpoint and will-retire alignment. Using our theorems is contingent upon showing the implementation satisfies the conditions in Section 3.

Sawada and Hunt [16] verified that a non-deterministic implementation with out-of-order retirement satisfies informed-flushpoint equality with a deterministic specification with no internal state (iFEND). Their verification strategy is to build an intermediate model with history variables, called the MAETT. From our result, they can now conclude that informed-pointwise flushing (IUND) also holds. In later work [17,18], they enhanced their implementation to support external interrupts, which led them to add non-determinism to their specification because of the problem of predicting how many instructions the implementation will have completed when an interrupt occurs. Because of the non-deterministic specification, we cannot conclude that pointwise flushing holds in this case.

Skakkebæk *et al.* [14,13] verify that a deterministic implementation with in-order retirement satisfies informed-flushpoint equality with a deterministic specification with no internal state (iFEDD). 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. Because of our result they are able to conclude informed-pointwise flushing (IUDD) holds.

Hosabettu, Srivas, and Gopalakrishnan [10,11,12,9] prove that a deterministic out-of-order implementation satisfies informed-flushpoint equality with a deterministic specification with no internal state. They first prove informed-pointwise flushing (IUDD), then apply induction to prove informed-flushpoint equality (iFEDD). Because they use IUDD as a step toward iFEDD, there is no need for our result in this work.

Arons and Pnueli [3] use flushpoint alignment, not informed-flushpoint. Thus, our result is not applicable to their verification effort.

Fox and Harman [7] use will-retire alignment for a deterministic implementation and specification where the match is projection of the implementation (WEDD). Based on the results of this paper, they can also conclude informed-pointwise equality (IEDD).

## 7 Conclusions

This paper contains three results. First, we prove that for deterministic specifications with no internal state, from multi-step informed-flushpoint equality, one can conclude single-step informed-pointwise with the flushing match. Second, we provide a counterexample showing that for non-deterministic specifications flushpoint equality does not always imply informed-pointwise with the flushing match. Third, we prove that a multi-step correctness statement based on synchronization at retirement with the equality match is equivalent to informed-pointwise with the equality match. Our results are applicable to superscalar implementations, which fetch or retire multiple instructions in a single step.

Our long-term goal in studying correctness statements abstractly is to determine decomposition strategies that will ease the verification effort. The proofs described in this paper have been mechanized in the HOL theorem prover. We have created a reusable theory of microprocessor correctness that allows the comparison and extension of existing verification efforts.

**Acknowledgments.** We thank Robert Jones of Intel and the reviewers for detailed comments on this paper. The authors are supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC). Aagaard is supported in part by Intel Corporation.

## References

1. M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones. A framework for microprocessor correctness statements. In *CHARME*, volume 2144 of *LNCS*, pages 433–448. Springer, 2001.
2. M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones. A framework for superscalar microprocessor correctness statements, 2002. To appear in *Software Tools for Technology Transfer*.
3. T. Arons and A. Pnueli. Verifying Tomasulo’s algorithm by refinement. In *Int’l Conf. on VLSI Design*, pages 92–99. IEEE Comp. Soc. Press, 1999.
4. S. Berezin, E. Clarke, A. Biere, and Y. Zhu. Verification of out-of-order processor designs using model checking and a light-weight completion function. *Formal Methods in System Design*, 20(2):159–186, March 2002.
5. J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, volume 818 of *LNCS*, pages 68–80. Springer, 1994.
6. N. A. Day, M. D. Aagaard, and M. Lou. A mechanized theory for microprocessor correctness statements. Technical Report 2002-11, U. of Waterloo, Dept. of Comp. Sci., 2002.
7. A. Fox and N. Harman. Algebraic models of correctness for microprocessors. *Formal Aspects in Computing*, 12(4):298–312, 2000.
8. M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
9. R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In *CAV*, volume 1855 of *LNCS*, pages 521–537. Springer, 2000.
10. R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In *CAV*, volume 1427 of *LNCS*, pages 122–134. Springer, 1998.
11. R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In *CAV*, volume 1633 of *LNCS*, pages 47–59. Springer, 1999.

12. R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor without reorder buffer using the completion functions approach. In *CHARME*, volume 1703 of *LNCS*, pages 8–22. Springer, 1999.
13. R. Jones, J. Skakkebæk, and D. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In *FMCAD*, volume 1522 of *LNCS*, pages 2–17. Springer, 1998.
14. R. B. Jones, J. U. Skakkebæk, , and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. *Formal Methods in System Design*, 20(2):39–58, March 2002.
15. R. Milner. An algebraic definition of simulation between programs. In *Joint Conference on Artificial Intelligence*, pages 481–489. British Computer Society, 1971.
16. J. Sawada and W. Hunt. Trace table based approach for pipelined microprocessor verification. In *CAV*, volume 1254 of *LNCS*, pages 364–375. Springer, 1997.
17. J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In *CAV*, volume 1427 of *LNCS*, pages 135–146. Springer, 1998.
18. J. Sawada and W. Hunt. Results of the verification of a complex pipelined machine model. In *CHARME*, volume 1703 of *LNCS*, pages 313–316. Springer, 1999.