A Mechanized Theory for Microprocessor Correctness Statements

Nancy A. Day[1], Mark D. Aagaard[2], and Meng Lou[1]
University of Waterloo Department of Computer Science Technical Report 2002-11

[1] Computer Science, University of Waterloo, Waterloo, ON, Canada
nday@cs.uwaterloo.ca, mlou@student.math.uwaterloo.ca
[2] Electrical and Computer Engr., University of Waterloo
markaa@swen.uwaterloo.ca

Abstract Microprocessor verification has become increasingly challenging with the use of optimizations such as out-of-order execution. Because of the complexity of the implementations, a wide variety of microprocessor correctness statements have been proposed and used in verification efforts. In this work, we have mechanized a previously proposed framework for classifying these correctness statements. We have verified the relationships between the different points in the framework, and developed a characterization of the commonly used flushing abstraction function. The relationships between points in the framework are general theorems that provide "verification highways" to connect different correctness statements and provide reusable verification strategies. We have used these highways to determine the precise relationships between top-level correctness statements used in verification efforts.

1 Introduction

The keystone of a microprocessor verification is the correctness statement, which describes the intended relationship between the implementation and specification. Recent advances in microprocessor complexity and verification technology have resulted in a wide variety of correctness statements. Comparisons between and extensions of verification efforts is difficult because of the diversity of correctness statements. Aagaard *et al.* [ACDJ01,ACDJ02] introduced a framework, which we call Microbox, to classify and compare simulation-style microprocessor correctness statements about safety – i.e. that any behaviour of the implementation is also a behaviour of the specification. They formally defined correctness statements in a common notation and proposed an informal ordering of the correctness statements based on notions of generality.

In this paper, we describe the formalization and proofs of the relationships between the different correctness statements of Microbox. These proofs have been mechanized in the HOL theorem proving system [GM93]. This process validated previous intuition in this regard but also pointed out some non-obvious relationships and assumptions. Formalizing the relationship between the correctness statements forced us to formalize requirements of several functions commonly used in microprocessor verification. Most notably, we have identified two conditions on the *flushing* abstraction function.

These two conditions allow us to separate the behaviour of flushing from the next-state function of the implementation.

Microbox was originally conceived as a way of understanding existing work. While working with Microbox, we hypothesized that it could provide “verification highways” that would bridge the gap from implementation-specific verification strategies to general notions of correctness. The mechanization of Microbox provides these verification highways. Our vision is that Microbox will now be useful both in choosing correctness statements and in structuring verification strategies. We illustrate this by stating and proving a previously unknown relationship between two top-level correctness statements used in the literature. One of these correctness statements, which we call flush-point alignment with an equality match, is the one used by Sawada and Hunt [SH97]. In this correctness statements, the projection of the external state of flushed implementation states is compared with specification states. The second correctness statement uses the flushing abstraction mechanism of Burch and Dill [BD94] and compares implementation traces in which only the last step of the trace fetches an instruction. This second correctness statement, which we call must-issue alignment with a flushing abstraction, is used by Berezin *et al.* [BBCZ98]. In this paper, we show that must-issue with the flushing match logically implies flush-point alignment with the equality match. This result relies on the two conditions characterizing the behaviour of the flushing abstraction function.

We begin with a description of the Microbox framework in Section 2. Section 3 states conditions on the behaviour of microprocessor-specific functions used in correctness statements. Sections 4 and 5 describe our proofs of the relationships between the elements within the framework. Section 6 outlines the proof of the relationship between two top-level correctness statements, demonstrating how the “verification highways” make it possible to relate verification efforts. Section 7 summarizes the paper and describes future directions.

2 The Microbox Framework

Formal verification of sequential microprocessors has generally been done using simulation-style correctness statements, where a step of the implementation is compared to a step of the specification. Pipelining and other optimizations increase the gap between the behaviour of the implementation and the specification, making it more difficult to consider only one step of the implementation and specification traces. Thus, the correctness statements have become more complex.

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 executions of the implementation and specification (Section 2.1). *Match* is the relation established between the aligned implementation and specification states (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.

By choosing options for the parameters, we arrive at a variety of correctness statements. The Microbox framework uses four-letter acronyms to describe the choice of option for each parameter: $\langle$*alignment*$\rangle$ $\langle$*match*$\rangle$ $\langle$*impl. execution*$\rangle$ $\langle$*spec. execution*$\rangle$. The options identified in Microbox for these parameters are listed in Table 1. For example, "IUDD" denotes informed-pointwise alignment (I), flushing match (U), and deterministic implementation (D) and specification (D). All of these correctness statements describe the induction step of the verification effort. We omit the base case as it is generally quite straightforward.

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

Table1. Options for correctness statement parameters

The alignment parameter determines the overall form of the induction clause, while the other parameters provide substitutions into the alignment definitions. The alignment options are described for a general relation match (O) and non-deterministic implementation (N) and specification machines (N) and therefore labelled by the alignment letter and ONN. Different matches are substitutions of the relation match. For deterministic machines, next state functions can be substituted for the next state relations.

In the Microbox framework, both the specification and implementation machines have program memories as part of their state, and so do not take instructions as inputs. We assume that 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. We use the following notation:

N is a next-state relation; $N^k(q, q')$ means q' is reachable from q in k steps of N.
Π is an external state projection function.
$q_i \stackrel{\Pi}{=} q_s$ says that q_i and q_s have equivalent external state: $\pi_i(q_i) = \pi_s(q_s)$.
The components are subscripted with "s" for specification and "i" for implementation.

In the following subsections, we describe the options for the alignment and match parameters identified by Aagaard *et al.* [ACDJ02]. This previous work proposed an ordering amongst the options for each parameter. It is this ordering that we have made precise and verified in this paper. 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. Therefore we omit further consideration of these last two parameters, and the results of this paper are all for non-deterministic specifications and implementations.

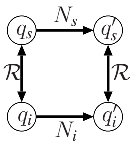
2.1 Alignment

Alignment describes which states in the execution trace are tested for matching. The following paragraphs briefly introduce five options for alignment.

Pointwise alignment (P) (Definition 1) is the classic commuting diagram. *Informed pointwise* (I) (Definition 2) is a variation of pointwise alignment that allows the implementation to inform the correctness statement as to whether it fetched an instruction, in which case the specification should take one step. If no instruction was fetched, then the specification should not take a step. This accommodates pipeline stalls. *Stuttering alignment* (S) (Definition 3) allows the specification to *stutter*, i.e. two or more consecutive implementation states can match the same specification state.

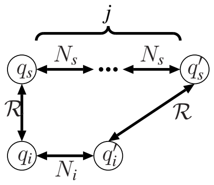
Definition 1 (Pointwise induction clause: PONN).

$$\mathsf{PONN}(\mathcal{R}, N_i, N_s) \equiv$$
$$\forall\, q_i, q_i'.\ \forall\, q_s.\ \ \exists\, q_s'.$$
$$\begin{bmatrix} \wedge\, N_i(q_i, q_i') \\ \mathcal{R}(q_i, q_s) \end{bmatrix} \implies \begin{bmatrix} \wedge\, N_s(q_s, q_s') \\ \mathcal{R}(q_i', q_s') \end{bmatrix}$$



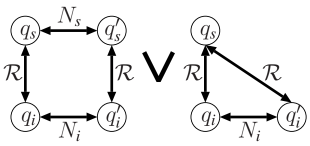
Definition 2 (Informed pointwise induction clause: IONN).

$$\mathsf{IONN}(\mathit{numFetch}, \mathcal{R}, N_i, N_s) \equiv$$
$$\forall\, q_i, q_i'.\ \forall\, q_s.\ \ \exists\, q_s'.$$
$$\text{let } j = \mathit{numFetch}(q_i, q_i') \text{ in}$$
$$\begin{bmatrix} \wedge\, N_i(q_i, q_i') \\ \mathcal{R}(q_i, q_s) \end{bmatrix} \implies \begin{bmatrix} \wedge\, N_s^j(q_s, q_s') \\ \mathcal{R}(q_i', q_s') \end{bmatrix}$$



Definition 3 (Stuttering induction clause: SONN).

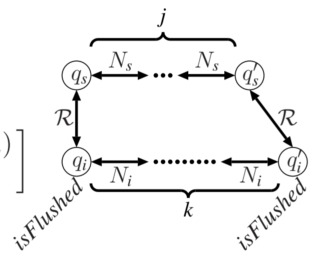
$$\mathsf{SONN}(\mathcal{R}, N_i, N_s) \equiv$$
$$\forall\, q_i, q_i'.\ \forall\, q_s.\ \ \exists\, q_s'.$$
$$\begin{bmatrix} \wedge\, N_i(q_i, q_i') \\ \mathcal{R}(q_i, q_s) \end{bmatrix} \implies \begin{bmatrix} \wedge\, (N_s(q_s, q_s')\ \vee\ (q_s' = q_s)) \\ \mathcal{R}(q_i', q_s') \end{bmatrix}$$



Flush-point alignment (Definition 4) (F) says that if there is a trace between flushed implementation states (i.e. no in-flight instructions), 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. Definition 4 says that if the implementation is in a flushed state q_i and can transition 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 transition through some number of steps j to a state q_s' that matches q_i'.

Definition 4 (Flush-point induction clause: FONN).

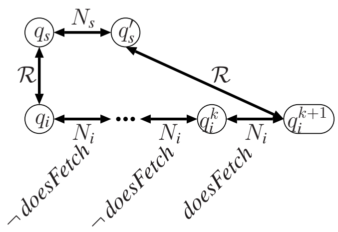
$$\mathsf{FONN}(\mathit{isFlushed}, \mathcal{R}, N_i, N_s) \equiv$$
$$\forall\, q_i, q_i', q_s.\ \exists\, q_s'.$$
$$\begin{bmatrix} \wedge\, \mathit{isFlushed}(q_i) \\ \wedge\, \exists\, k.\ N_i^k(q_i, q_i') \\ \wedge\, \mathit{isFlushed}(q_i') \\ \mathcal{R}(q_i, q_s) \end{bmatrix} \implies \begin{bmatrix} \wedge\, \exists\, j.\ N_s^j(q_s, q_s') \\ \mathcal{R}(q_i', q_s') \end{bmatrix}$$



In *must-issue alignment* (M) (Definition 5), the specification takes one step. The implementation takes steps until it reaches a state where it can fetch an instruction, followed by one more step from q_i^k to q_i^{k+1} where it fetches an instruction.

Definition 5 (Must-issue induction clause: MONN).

$$
\begin{array}{l}
\mathsf{MONN}(\mathit{doesFetch}, \mathcal{R}, N_i, N_s) \equiv \\
\quad \forall\, q_i^0, q_i^1, \ldots, q_i^{k+1}.\ \forall\, q_s.\ \exists\, q_s'. \\
\qquad \left[\begin{array}{l}
\quad (\forall\, j < k.\ N_i(q_i^j, q_i^{j+1}) \wedge \\
\quad \neg\, \mathit{doesFetch}(q_i^j, q_i^{j+1})) \\
\wedge\ N_i(q_i^k, q_i^{k+1}) \\
\wedge\ \mathit{doesFetch}(q_i^k, q_i^{k+1}) \\
\wedge\ \mathcal{R}(q_i^0, q_s)
\end{array} \right] \implies \left[\begin{array}{l}
\quad N_s(q_s, q_s') \\
\wedge\ \mathcal{R}(q_i^{k+1}, q_s')
\end{array} \right]
\end{array}
$$



Aagaard *et al.* [ACDJ02] proposed that an ordering among these alignment options based on generality is pointwise (P), informed pointwise (I), stuttering (S), and flush-point (F). We prove this order is logical implication with some side conditions in Section 4. Characterizing the relationship between must-issue and the others was more challenging, and depends on the match parameter chosen (Section 6).

In the presentation of Microbox given in [ACDJ02], the correctness statements P, S, and I are generalized to superscalar machines of arbitrary width. In our proofs, we worked with the singlescalar correctness statements, but do not anticipate any difficulties in generalizing to superscalar.

2.2 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 1 shows the matches identified by the Microbox framework as used in microprocessor verification. These options are substituted in for $\mathcal{R}$ in an alignment option to create a correctness statement. The arrows show the partial order proposed by Aagaard *et al.* [ACDJ02], where definitions lower in the order are instances of higher options. Our verification of these relationships is described in Section 5.

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 point 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) uses an abstraction function that preserves the externally-visible part of the implementation state. 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 $\Pi(q_i) = q_s$, depicted by the dashed line connecting E and R in Figure 1.

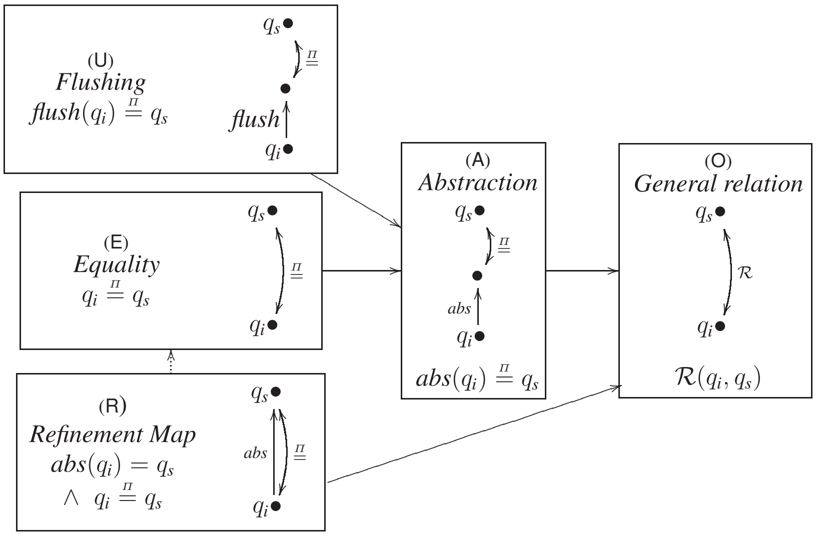


Figure1. Options and partial order for the match parameter

2.3 Correctness Space

The classification of correctness statements in Microbox together with the relationships proposed among the points in the framework allow us to map out the space of correctness statements in the partial order of Figure 2. For clarity we have omitted the NN suffix on all the points. Options for the alignment parameter run up the vertical axis and the options for the match parameter span the horizontal axis. Aagaard *et al.* [ACDJ02] show where most verification efforts described in the literature fit on this map.

In the following sections we describe our verification of this map. We show that IUNN and MUNN are equivalent under the conditions given in Section 3. Also FENN and FUNN are equivalent. Our proofs create "verification highways" that allow us to precisely describe the relationships between top-level correctness statements. In Section 6, we use these highways to prove that MUNN implies FENN.

3 Characterization of Microprocessor-Specific Functions

The proofs described in the following sections relating the points in the Microbox framework depend on certain information about the microprocessor-specific functions used in the definitions of the framework points. In order to remain general for different implementations, we describe the minimum amount of information necessary as conditions on these functions. All of the following conditions are reasonable for modern microprocessor implementations. These conditions were discovered in carrying out the proofs of the theorems of the next section, but form a general specification of the behaviour of these functions for particular implementations.

The microprocessor-specific functions used in the correctness statements are:

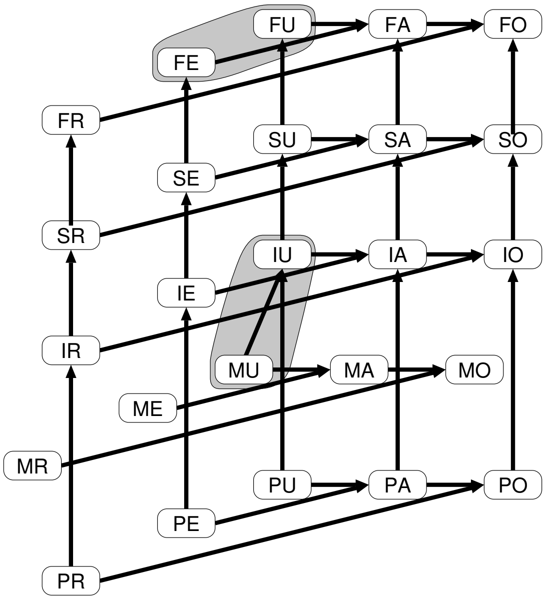


Figure2. Space of correctness statements

doesFetch(q_i, q_i') – true if an instruction is fetched in the step from q_i to q_i'.
numFetch(q_i, q_i') – returns the number of instructions fetched in a step. In a deterministic implementation, this function usually depends only on q_i.
flush(q_i) – produces an implementation state with no in-flight instructions.
isFlushed(q_i) – true if a state is flushed.

We characterize the relationship amongst these functions using Conditions 1–4.

Our first condition relates the behaviour of *doesFetch* and *numFetch*. In a singlescalar implementation, these two functions are different ways of stating the same information about whether an implementation step fetches an instruction. We need both of them because they are used in different correctness statements. Condition 1 states that if *doesFetch* is true of a step then *numFetch* is one, otherwise it is zero.

Condition 1 (*doesFetch* and *numFetch*)

$$\begin{array}{l} \mathit{doesfetch\_cond}(\mathit{numFetch}, \mathit{doesFetch}) \equiv \\ \quad \forall\, q_i, q_i'.\ \mathit{numFetch}(q_i, q_i') = \text{if } \mathit{doesFetch}(q_i, q_i') \text{ then } 1 \text{ else } 0 \end{array}$$

We characterize the required behaviour of a flushing function with two conditions. Condition 2 relates the function *flush* to the predicate *isFlushed* and says that if a state q is flushed, then flushing q returns q.

Condition 2 (*isFlushed* and *flush*)

$$\mathit{fl\_no\_effect\_cond}(\mathit{isFlushed}, \mathit{flush}) \equiv \forall\, q.\ \mathit{isFlushed}(q) \implies \mathit{flush}(q) = q$$

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*)

$$\begin{array}{l} \mathit{fl\_no\_fetch\_cond}(\mathit{doesFetch}, \mathit{flush}) \equiv \\ \quad \forall\, q, q'.\, \neg\, \mathit{doesFetch}(q, q') \wedge N(q, q') \implies \mathit{flush}(q) = \mathit{flush}(q') \end{array}$$

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 [BD94]. 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.* [HSG98], who write completion functions for each stage in the pipeline and then compose the completion functions to create a flushing function.

The final condition we need in our proofs is the restriction that we are, for the moment, only handling singlescalar machines. This condition is needed to relate the singlescalar versions of IONN and SONN. We expect the generalization to superscalar machines to be straightforward and we could then eliminate the need for Condition 4.

Condition 4 (Singlescalar)

$$\begin{array}{l} \mathit{singlescalar\_cond}(\mathit{numFetch}) \equiv \\ \quad \forall\, q, q'.\, (\mathit{numFetch}(q, q') = 0) \vee (\mathit{numFetch}(q, q') = 1) \end{array}$$

4 Verification of Alignment Ordering

Aagaard *et al.* [ACDJ02] ordered the options for the alignment parameter based on "generality". We have formally proven that four of the five options for the alignment parameter with the most general match can be ordered using logical implication. Theorems for the alternative options (A, U, E, R) are derived by substituting the appropriate match option into the general theorems. The fifth option, M (must-issue), fits in the order only for the flushing match and is the focus of Section 6.

4.1 Pointwise to Informed Pointwise

Pointwise alignment (PONN — Definition 1) implies informed pointwise alignment (IONN — Definition 2) under the condition that the function *numFetch* returns one for each step of the implementation (Theorem 1). This restriction is not surprising, because informed pointwise is a more general case of pointwise that accommodates implementations that can stall.

Theorem 1 (Pointwise implies informed pointwise).

$$\begin{array}{l} \forall\, \mathcal{R}, N_i, N_s, \mathit{numFetch}\,. \\ \quad (\forall\, qi, qi'.\ \mathit{numFetch}(qi, qi') = 1) \implies \\ \qquad \mathsf{PONN}(\mathcal{R}, N_i, N_s) \implies \mathsf{IONN}(\mathit{numFetch}, \mathcal{R}, N_i, N_s) \end{array}$$

4.2 Informed Pointwise to Stuttering

Informed pointwise (IONN — Definition 2) uses *numFetch* to say whether the specification should take a step. Stuttering alignment (SONN — Definition 3) allows a step of the implementation to match either a step of the specification, or the current specification state. To show that informed pointwise implies stuttering, we only need the singlescalar condition, which states that *numFetch* always returns one or zero.

Theorem 2 (Informed pointwise implies stuttering).

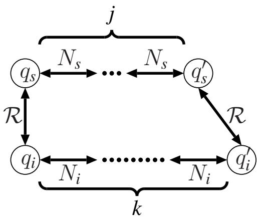
$$\begin{array}{l} \forall\, \mathcal{R}, N_i, N_s, \mathit{numFetch} \\ \quad \mathit{singlescalar\_cond}(\mathit{numFetch}) \implies \\ \qquad \mathsf{IONN}(\mathit{numFetch}, \mathcal{R}, N_i, N_s) \implies \mathsf{SONN}(\mathcal{R}, N_i, N_s) \end{array}$$

4.3 Stuttering to Flush-Point

We proved that stuttering alignment (SONN — Definition 3) implies flush-point alignment (FONN — Definition 4) by introducing an intermediate correctness statement that we call multi-step stuttering (msSONN — Definition 6). In multi-step stuttering, for any implementation trace of length k from q_i to q_i', and all specification states q_s that match q_i under $\mathcal{R}$, there exists a specification trace of length j from q_s that leads to a specification state, q_s', that matches q_i' under $\mathcal{R}$. We used induction over the trace length to prove that stuttering alignment implies multi-step stuttering alignment (Theorem 3).

Definition 6 (Multi-step stuttering induction clause: msSONN).

$$\begin{array}{l} \mathsf{msSONN}(\mathcal{R}, N_i, N_s) \equiv \\ \quad \forall\, q_i, q_i'.\ \forall\, q_s.\ \exists\, q_s'. \\ \quad \left[\wedge \begin{array}{l} \exists\, k.\ N_i^k(q_i, q_i') \\ \mathcal{R}(q_i, q_s) \end{array}\right] \implies \left[\wedge \begin{array}{l} \exists\, j.\ N_s^j(q_s, q_s') \\ \mathcal{R}(q_i', q_s') \end{array}\right] \end{array}$$



Theorem 3 (Stuttering implies multi-step stuttering).

$$\begin{array}{l} \forall\, \mathcal{R}, N_i, N_s. \\ \quad \mathsf{SONN}(\mathcal{R}, N_i, N_s) \implies \mathsf{msSONN}(\mathcal{R}, N_i, N_s) \end{array}$$

The identification of multi-step stuttering alignment was a critical step in verifying the relationship between the stuttering and flush-point alignment options, but we have not seen it used in practice. Flush-point alignment is just multi-step stuttering limited to checking only implementation traces that start and end in states that satisfy the *isFlushed* predicate (Theorem 4).

Theorem 4 (Multi-step stuttering implies flush-point).

$$\begin{array}{l} \forall\ \mathit{isFlushed}, \mathcal{R}, N_i, N_s. \\ \quad \mathsf{msSONN}(\mathcal{R}, N_i, N_s) \implies \mathsf{FONN}(\mathit{isFlushed}, \mathcal{R}, N_i, N_s) \end{array}$$

We used Theorems 3 and 4 to prove Theorem 5.

Theorem 5 (Stuttering implies flush-point).

$$\begin{array}{l} \forall\ \mathit{isFlushed}, \mathcal{R}, N_i, N_s. \\ \quad \mathsf{SONN}(\mathcal{R}, N_i, N_s) \implies \mathsf{FONN}(\mathit{isFlushed}, \mathcal{R}, N_i, N_s) \end{array}$$

5 Verification of Match Ordering

The partial order of the matching options (Figure 1) is based on implication with some instantiation. Identifiers such as $\mathcal{R}$ and *abs* are intended to be substituted with specific relations or abstraction functions in a correctness statement. In this section we show the most general instance of each theorem: flush-point alignment with non-deterministic implementation and specification. We have also verified each theorem for the other alignment and execution instances.

5.1 Abstraction to Other

Abstracting the implementation state and testing for externally visible equality is a relation between an implementation state and a specification state. As such, abstraction is an instance of the most general match (Theorem 6).

Theorem 6 (Abstraction is other with abstraction).

$$\begin{array}{l} \forall\ \mathit{isFlushed}, \mathit{abs}, \pi_i, \pi_s, \mathcal{R}, N_i, N_s. \\ \quad \mathcal{R} = (\lambda\ (q_i, q_s).\ \pi_s(q_s) = \pi_i(\mathit{abs}(q_i))) \implies \\ \quad\quad \mathsf{FANN}(\mathit{isFlushed}, \mathit{abs}, \pi_i, \pi_s, N_i, N_s) \iff \mathsf{FONN}(\mathit{isFlushed}, \mathcal{R}, N_i, N_s) \end{array}$$

5.2 Flushing to Abstraction

Theorem 7 says that flushing is a special form of abstraction.

Theorem 7 (Flushing is abstraction with flush).

$$\begin{array}{l} \forall\ \mathit{isFlushed}, \mathit{flush}, \mathit{abs}, \pi_i, \pi_s, N_i, N_s. \\ \quad \mathit{flush} = \mathit{abs} \implies \\ \quad\quad \mathsf{FUNN}(\mathit{isFlushed}, \mathit{flush}, \pi_i, \pi_s, N_i, N_s) \iff \mathsf{FANN}(\mathit{isFlushed}, \mathit{abs}, \pi_i, \pi_s, N_i, N_s) \end{array}$$

5.3 Equality to Abstraction

The equality match says that the externally visible state components are equal. Theorem 8 says that equality match is equivalent to abstraction where the abstraction function is identity.

Theorem 8 (Equality is abstraction with identity).

$$\begin{array}{l} \forall\ \mathit{isFlushed}, \mathit{abs}, \pi_i, \pi_s, N_i, N_s. \\ \quad \mathit{abs} = (\lambda q.\ q) \implies \\ \quad\quad \mathsf{FENN}(\mathit{isFlushed}, \pi_i, \pi_s, N_i, N_s) \iff \mathsf{FANN}(\mathit{isFlushed}, \mathit{abs}, \pi_i, \pi_s, N_i, N_s) \end{array}$$

5.4 Refinement to Other

Theorem 9 says that a verification of flush-point refinement is equivalent to a flush-point other verification with the relation $\mathcal{R}$ being that the specification state is a refinement of the implementation state. As discussed in Section 5.5, a refinement match does not necessarily imply an equality match or an abstraction match.

Theorem 9 (Refinement is other with refinement as R).

$$\begin{array}{l} \forall\ \mathit{isFlushed}, \mathit{abs}, \pi_i, \pi_s, \mathcal{R}, N_i, N_s. \\ \quad \mathcal{R} = (\lambda\ (q_i, q_s).\ (\pi_i(q_i) = \pi_s(q_s)) \wedge (q_s = \mathit{abs}(q_i))) \implies \\ \qquad \mathsf{FRNN}(\mathit{isFlushed}, \mathit{abs}, \pi_i, \pi_s, N_i, N_s) \iff \mathsf{FONN}(\mathit{isFlushed}, \mathcal{R}, N_i, N_s) \end{array}$$

5.5 Special Cases

There are a number of special cases in the ordering of matching relations. First, we explain why flush-point alignment with equality match is the same as flush-point alignment with flushing match. Second, we explain why the refinement match does not imply the equality match for commuting diagrams.

Flush-point Equality is Flush-Point Flushing Flush-point alignment with a flushing match is equivalent to flush-point alignment with equality match because, from Condition 2, flushing a flushed state results in the same state (Theorem 10). A similar theorem was proved by Hosabettu *et al.* [HSG98].

Theorem 10 (Flush-point flushing is flush-point equality).

$$\begin{array}{l} \forall\ \mathit{isFlushed}, \mathit{flush}, \pi_i, \pi_s, N_i, N_s. \\ \quad \mathit{fl\_no\_effect\_cond}(\mathit{isFlushed}, \mathit{flush}) \implies \\ \qquad \mathsf{FUNN}(\mathit{isFlushed}, \mathit{flush}, \pi_i, \pi_s, N_i, N_s) \iff \mathsf{FENN}(\mathit{isFlushed}, \pi_i, \pi_s, N_i, N_s) \end{array}$$

Why Refinement Does not Imply Equality In general, the refinement match does not imply the equality match. Refinement implies equality only if the specification has no internal state (the projection function π_s is the identity function), in which case, the two correctness statements are equivalent.

The lack of a relationship between equality and refinement might be surprising, and indeed, we tried several times to prove that refinement is stronger than equality. But, in hindsight, the problem is clear. To prove a commuting diagram (the induction step of a correctness statement), we assume that the match holds on the current states q_i and q_s and then prove that it holds in the resulting states q_i' and q_s'. As illustrated in Figure 3, the equality match is insufficient to discharge the antecedent of the assumption for the refinement match. This problem is independent of the alignment used. Refinement match does not imply abstraction match or flushing match for the same reason that refinement does not imply equality.

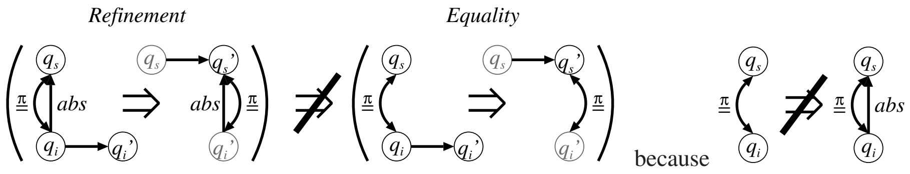


Figure3. Refinement does not imply equality

Our intuition that refinement is stronger than equality was based on considering traces, rather than individual steps of the implementation and specification. As part of our future work, we are extending the framework to connect the commuting diagrams to notions of trace containment. At the level of traces we expect to verify a relationship between refinement and equality.

6 Must-Issue Flushing and Flush-Point Equality

In this section, we describe the relationship between the must-issue parameter and other points of the framework. The MONN correctness statement (Definition 5) compares one specification step with an implementation trace that fetches only one instruction. Starting from any implementation state, the implementation may take a number of steps where it is unable to fetch an instruction before fetching an instruction. Thus, the implementation trace is dependent on a *doesFetch* predicate that is true if an instruction is fetched in an implementation step. With a general relation as the match, we have not found that must-issue relates to any of the other points in the framework. Also, as reported in Aagaard *et al.* [ACDJ02], we have not found the general case of MONN used as a correctness statement in any verification efforts.

However, must-issue alignment with the flushing abstraction match (MUNN — Definition 7) is used in verification efforts. For example, Berezin *et al.* [BBCZ98] use MUDD, which is must-issue with flushing and deterministic specification and implementation machines as their top-level correctness statement.

Definition 7 (Must-issue with flushing match induction clause: MUNN).

$$
\begin{array}{l}
\mathsf{MUNN}(\mathit{doesFetch}, \mathit{flush}, \pi_i, \pi_s, N_i, N_s) \equiv \\
\quad \forall\, q_i^0, q_i^1, \ldots, q_i^{k+1}.\ \forall\, q_s.\ \exists\, q_s'. \\
\qquad \left[\begin{array}{l}
(\forall\, j < k.\ N_i(q_i^j, q_i^{j+1}) \wedge \\
\qquad \neg\, \mathit{doesFetch}(q_i^j, q_i^{j+1})) \\
\wedge\ N_i(q_i^k, q_i^{k+1}) \\
\wedge\ \mathit{doesFetch}(q_i^k, q_i^{k+1}) \\
\wedge\ \pi_i(\mathit{flush}(q_i^0)) = \pi_s(q_s)
\end{array}\right]
\Longrightarrow
\left[\begin{array}{l}
N_s(q_s, q_s') \\
\wedge\ \pi_i(\mathit{flush}(q_i^{k+1})) = \pi_s(q_s')
\end{array}\right]
\end{array}
$$

Because of the required behaviour of the *flush* function, we are able to connect MUNN with the rest of the framework. Once connected, we can use the verification highways to show MUNN implies FENN (Definition 8), another common top-level correctness statement used in verification efforts. The proof is done with six theorems and transitivity as illustrated in Figure 4, where the lighter lines are instances of the bold lines on the right of the picture.

Definition 8 (Flush-point with equality induction clause: FENN).

$$
\begin{array}{l}
\mathsf{FENN}(\mathit{isFlushed}, \pi_i, \pi_s, N_i, N_s) \equiv \\
\quad \forall\, q_i, q_i', q_s.\ \exists\, q_s'. \\
\qquad \left[\begin{array}{l}
\mathit{isFlushed}(q_i) \\
\wedge\ \exists\, k.\ N_i^k(q_i, q_i') \\
\wedge\ \mathit{isFlushed}(q_i') \\
\wedge\ \pi_i(q_i) = \pi_s(q_s)
\end{array}\right]
\Longrightarrow
\left[\begin{array}{l}
\exists\, j.\ N_s^j(q_s, q_s') \\
\wedge\ \pi_i(q_i') = \pi_s(q_s')
\end{array}\right]
\end{array}
$$

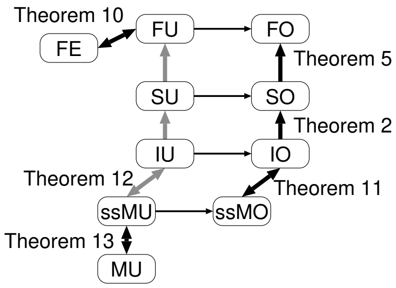


Figure4. Verification path for must-issue flushing implies flush-point equality

The first insight in this proof was the introduction of an alternative way of expressing MUNN that appears to be a tighter correctness criteria, but under the assumed behaviour of the *flush* function, is actually equivalent to MUNN.

We introduced a new correctness statement, which we call single-step must-issue (ssMONN – Definition 9). We proved that ssMUNN (ssMONN with the flushing match) is equivalent to MUNN. ssMONN decomposes MONN into two simpler, single-step, properties based on whether the implementation will fetch an instruction. ssMONN is similar to informed pointwise, except that instead of having the knowledge of how many instructions are fetched in an implementation step, the predicate *doesFetch* is used to indicate whether an instruction is fetched in the implementation step.

Definition 9 (Single-step must issue induction clause: ssMONN).

$$
\begin{array}{l}
\mathsf{ssMONN}(\mathit{flush}, \mathit{doesFetch}, \mathcal{R}, N_i, N_s) \equiv \\
\quad \forall\, q_i, q_i', q_s. \\
\qquad \mathit{doesFetch}(q_i, q_i') \implies \left[\begin{array}{l} \exists\, q_s'. \\ \quad \left[\wedge \begin{array}{l} N_i(q_i, q_i') \\ \mathcal{R}(q_i, q_s) \end{array}\right] \implies \left[\wedge \begin{array}{l} N_s(q_s, q_s') \\ \mathcal{R}(q_i', q_s') \end{array}\right] \end{array}\right] \\
\quad \wedge \\
\qquad \neg\, \mathit{doesFetch}(q_i, q_i') \implies \left[\quad \left[\wedge \begin{array}{l} N_i(q_i, q_i') \\ \mathcal{R}(q_i, q_s) \end{array}\right] \implies [\mathcal{R}(q_i', q_s)] \right]
\end{array}
$$

We proved that single-step must-issue is equivalent to informed pointwise alignment under Condition 1, the *doesfetch_cond*, which relates the predicate *doesFetch* with the function *numFetch* (Theorem 11).

Theorem 11 (Informed pointwise is single-step must-issue).

$$
\begin{array}{l}
\forall\ \mathit{numFetch}, \mathit{doesFetch}, \mathcal{R}, N_i, N_s. \\
\quad \mathit{doesfetch\_cond}(\mathit{numFetch}, \mathit{doesFetch}) \implies \\
\qquad (\mathsf{IONN}(\mathit{numFetch}, \mathcal{R}, N_i, N_s) \iff \mathsf{ssMONN}(\mathit{doesFetch}, \mathcal{R}, N_i, N_s))
\end{array}
$$

By specializing $\mathcal{R}$ in Theorem 11 with the flush match, we were able to conclude that IUNN is equivalent to ssMUNN (Theorem 12).

Theorem 12 (Informed pointwise with flushing match is single-step must-issue with flushing match).

$$\begin{array}{l} \forall\ \mathit{doesFetch}, \mathit{numFetch}, \mathit{flush}, \pi_i, \pi_s, N_i, N_s. \\ \mathit{doesfetch\_cond}(\mathit{numFetch}, \mathit{doesFetch}) \implies \\ \mathsf{IUNN}(\mathit{numFetch}, \mathit{flush}, \pi_i, \pi_s, N_i, N_s) \iff \mathsf{ssMUNN}(\mathit{doesFetch}, \mathit{flush}, \pi_i, \pi_s, N_i, N_s) \end{array}$$

Next, we proved that single-step must-issue (ssMUNN) and must-issue (MUNN) are equivalent for the flushing match. The proof that ssMUNN implies MUNN was a straightforward application of induction over the number of implementation steps in MUNN and a case split on whether the added step in the trace would fetch an instruction. In fact, this direction of the proof holds for the general matching relation of "other" (O). The proof that MUNN implies ssMUNN required Condition 3, which states that if an instruction is not fetched in a step from q_i to q_i' then the flush function results in the same state for both q_i and q_i'. This is illustrated in Figure 5 and stated in Theorem 13. The lighter lines in MUNN in Figure 5 show Condition 3.

Putting together Theorems 12 and 13, we are able to conclude that MUNN implies IUNN as long as Conditions 1 (relating *doesFetch* and *numFetch*) and 2 (relating *doesFetch* and *flush*) hold.

Theorem 13 (Single-step must-issue with flush match is multi-step must issue with flush match).

$$\begin{array}{l} \forall\ \mathit{doesFetch}, \mathit{flush}, \pi_i, \pi_s, N_i, N_s. \\ \quad \mathit{fl\_no\_effect\_cond}(\mathit{doesFetch}, \mathit{flush}) \implies \\ \quad\quad \mathsf{ssMUNN}(\mathit{doesFetch}, \mathit{flush}, \pi_i, \pi_s, N_i, N_s) \iff \\ \quad\quad\quad \mathsf{MUNN}(\mathit{doesFetch}, \mathit{flush}, \pi_i, \pi_s, N_i, N_s) \end{array}$$

At this point, we were able to ride a verification highway to connect MUNN to FENN. Showing IUNN implies FUNN can be done by transitivity using the flush match instances of IONN, SONN, and FONN, and the implications relationships of Theorem 2 (IONN implies SONN) and Theorem 5 (SONN implies FONN). FUNN and FENN are equivalent as given by Theorem 10. Therefore, we are able to conclude that any implementation and specification that satisfy the must-issue with flushing correctness statement (MUNN), where the flushing abstraction function obeys our four conditions, will also satisfy the flush-point with equality correctness statement (FENN) (Theorem 14).

The result that MUNN implies FENN states precisely the previously unknown result about the logical relationship between these two top-level correctness statements. Thus from Berezin *et al.*'s [BBCZ98] proof of MUDD for a microprocessor, they can also conclude FEDD holds for that microprocessor, thereby making it possible to precisely relate their work to that of Sawada and Hunt [SH97].

Theorem 14 (Must-issue with flushing match implies flush-point with equality match).

$$\begin{array}{l} \forall\ \mathit{flush}, \pi_i, \pi_s, n_i, n_s, \mathit{doesFetch}, \mathit{isFlushed}\,. \\ \left[\begin{array}{l} \mathit{doesfetch\_cond}(\mathit{numFetch}, \mathit{doesFetch}) \\ \wedge\ \mathit{fl\_no\_effect\_cond}(\mathit{isFlushed}, \mathit{flush}) \\ \wedge\ \mathit{fl\_no\_fetch\_cond}(\mathit{doesFetch}, \mathit{flush}) \\ \wedge\ \mathit{singlescalar\_cond}(\mathit{numFetch}) \end{array}\right] \implies \left[\begin{array}{c} \mathsf{MUNN}(\mathit{abs}, \pi_i, \pi_s, n_i, n_s, \mathit{doesFetch}) \\ \implies \\ \mathsf{FENN}(\pi_i, \pi_s, n_i, n_s, \mathit{isFlushed}) \end{array}\right] \end{array}$$

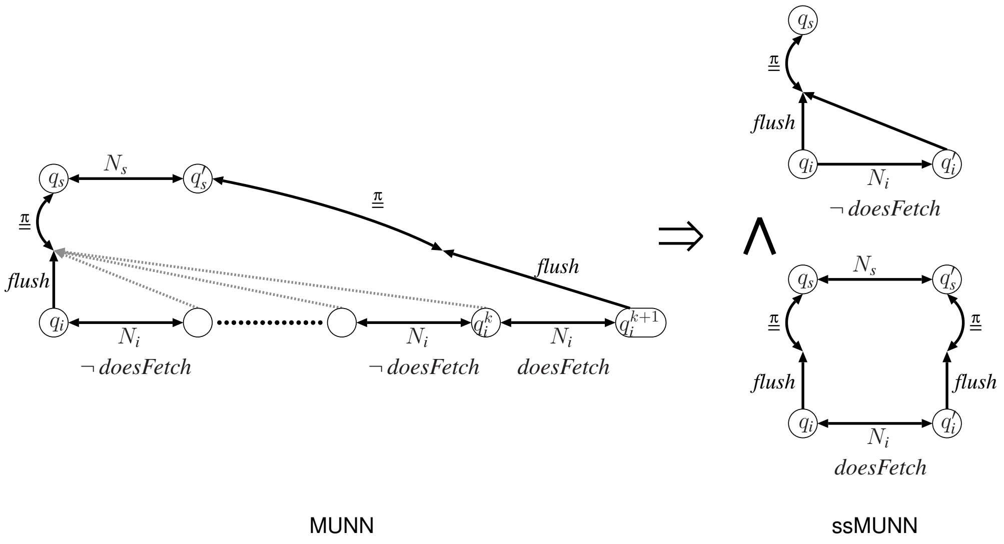


Figure5. Must-issue implies single-step must-issue showing ¬ *doesFetch* and *doesFetch* cases

7 Conclusions and Future Work

In this paper, we describe the verification of the precise relationships between correctness statements in the Microbox framework. These proofs depend on several quite reasonable conditions relating the microprocessor-specific functions used in the correctness statements. These conditions are stated generally and therefore should be applicable to most implementations.

By verifying these relationships, we have created "verification highways" for proofs of microprocessor correctness. Once a verification effort is associated with one point in the correctness space, it is possible to follow the highways to determine the relationship to correctness statements used in other verification efforts. We demonstrated the use of these highways to determine the previously unknown relationship between two top-level correctness statements, namely that must-issue alignment with the flushing match logically implies flush-point alignment with the equality match.

The proofs described in this paper have been mechanized in the HOL theorem proving system [GM93]. We mechanized each point in the framework and then verified its relationship (both horizontal and vertical connections) to the points around it. Transitivity links the points creating the verification highways. We expect the verification could easily be repeated in other theorem proving systems.

There are several directions for future research. We still have to determine the relationship of the "will-retire" alignment option identified in Microbox to the other points in the correctness space. We would like to further clarify the relationship between IUNN (informed pointwise with flushing match), and FENN (flush-point alignment with equality match). From this paper, we can conclude IUNN implies FENN. It is possible the implication can be reversed following a different line of reasoning than through stuttering alignment. Flush-point is less strict than stuttering because it only considers traces that start and end in flushed states. Sawada and Hunt [SH97] were able to prove IUNN and FENN for an implementation so it is possible that flush-point alignment is just a convenience, not a necessity. We would also like to relate these simulation-style correctness statements to trace-containment-style correctness statements. This relationship is clear for pointwise alignment, but less clear for the other alignment parameters. Finally, we want to connect our mechanized framework to verification strategies to create common on-ramps to our microprocessor verification highways.

Acknowledgments

We thank Byron Cook of Prover Technologies, and Robert Jones of Intel for early discussions on this topic. Some figures and equations were taken from Aagaard *et al.* [ACDJ02]. 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

[ACDJ01] 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.

[ACDJ02] 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*.

[BBCZ98] S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *FMCAD*, volume 1522 of *LNCS*, pages 369–386. Springer Verlag; New York, 1998.

[BD94] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, volume 818 of *LNCS*, pages 68–80. Springer Verlag; New York, 1994.

[GM93] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[HSG98] 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 Verlag; New York, 1998.

[SH97] J. Sawada and W. Hunt. Trace table based approach for pipelined microprocessor verification. In *CAV*, volume 1254 of *LNCS*, pages 364–375. Springer Verlag; New York, 1997.