Proof of Correctness of a Processor with Reorder Buffer Using the Completion Functions Approach [⋆]

Ravi Hosabettu[1], Mandayam Srivas[2], and Ganesh Gopalakrishnan[1]

[1] Department of Computer Science, University of Utah, Salt Lake City, UT 84112, hosabett,ganesh@cs.utah.edu

[2] Computer Science Laboratory, SRI International, Menlo Park, CA 94025, srivas@csl.sri.com

Abstract. The *Completion Functions Approach* was proposed in [HSG98] as a systematic way to decompose the proof of correctness of pipelined microprocessors. The central idea is to construct the abstraction function using completion functions, one per unfinished instruction, each of which specifies the effect (on the observables) of completing the instruction. In this paper, we show that this "instruction-centric" view of the completion functions approach leads to an elegant decomposition of the proof for an out-of-order execution processor with a reorder buffer. The proof does not involve the construction of an explicit intermediate abstraction, makes heavy use of strategies based on decision procedures and rewriting, and addresses both safety and liveness issues with a clean separation between them.

1 Introduction

For formal verification to be successful in practice not only is it important to raise the level of automation provided but is also essential to develop methodologies that scale verification to large state-of-the-art designs. One of the reasons for the relative popularity of model checking in industry is that it is automatic when readily applicable. A technology originating from the theorem proving domain that can potentially provide a similarly high degree of automation is one that makes heavy use of decision procedures for the combined theory of boolean expressions with uninterpreted functions and linear arithmetic [CRSS94,BDL96]. Just as model checking suffers from a state-explosion problem, a verification strategy based on decision procedures suffers from a "case-explosion" problem. That is, when applied naively, the sizes of the terms generated and the number of examined cases during validity checking explodes. Just as compositional model checking provides a way of decomposing the overall proof and reducing the effort for an individual model checker run, a practical methodology for decision

[⋆] This work was done in part when the first author was visiting SRI International in summer 1998. The work done by the authors at University of Utah was supported in part by NSF through Grant no. CCR-9800928. The work done by the authors at SRI International was supported in part by NASA contract NAS1-20334 and ARPA contract NASA-NAG-2-891 (ARPA Order A721).

procedure-centered verification must prescribe a systematic way to decompose the correctness assertion into smaller problems that the decision procedures can handle.

In [HSG98], we proposed such a methodology for pipelined processor verification called the *Completion Functions Approach.* The central idea behind this approach is to define the abstraction function as a composition of a sequence of completion functions, one for every unfinished instruction, in their program order. A completion function specifies how a partially executed instruction is to be completed in an atomic fashion, that is, desired effect on the observables of completing that instruction. Given such a definition of the abstraction function in terms of completion functions, the methodology prescribes a way of organizing the verification into proving a hierarchy of *verification conditions.* The methodology has the following attributes:

- The verification proceeds incrementally making debugging and error tracing easier.
- The verification conditions and most of the supporting lemmas needed to support the incremental methodology can be generated systematically.
- Every generated verification condition and lemma can be proved, often automatically, using a strategy based on decision procedures and rewriting.

In summary, the completion functions approach strikes a balance between full automation that (if at all possible) can potentially overwhelm the decision procedures, and a potentially tedious manual proof. This methodology is implemented using PVS [ORSvH95] and was applied (in [HSG98]) to three processor examples: DLX [HP90], dual-issue DLX, and a processor that exhibited limited out-of-order execution capability. An attribute common to all these processors was that the maximum number of instructions pending at any time in the pipeline was small and fixed, which made the completion functions approach readily amenable for these examples. It was an open question if the approach would be practical, even if applicable, to verify a truly out-of-order execution processor with a reorder buffer. Such a processor can have scores of pending instructions in the reorder buffer potentially making the task of defining completion functions tedious and possibly exploding the number of generated verification conditions.

In this paper, we demonstrate that the completion functions approach is well-suited to the verification of out-of-order execution processors by verifying an example processor (a simplified model, based on the P6 design) with a reorder buffer and generic execution units and without any data size bounds. We observe that regardless of how many instructions are pending in the reorder buffer, the instructions can only be in one of four distinct states. We exploit this fact to provide a single compact parameterized completion function applicable to all the pending instructions in the buffer. The abstraction function is then defined as a simple recursive function that completes all the pending instructions in the order in which they are stored in the reorder buffer. The proof is organized as a single parameterized verification condition, which is proved using a simple induction on the number of instructions in the buffer. The different cases of the induction are generated on the basis of how an instruction makes a transition from its present state to its next state. We make heavy use of an automatic case-analysis strategy and certain other strategies based on decision procedures and rewriting in discharging these different cases. This same observation about instruction state transitions is used in providing a proof of liveness too.

Related work: The problem of verifying the control logic of out-of-order execution processors has received considerable attention in the last couple of years using both theorem proving and model checking approaches. The following yardsticks can be used to evaluate the various approaches: (1) the amount and complexity of information required from the user, (2) the complexity of the manual steps of the methodology (3) the level of automation with which the obligations generated by the methodology can be verified.

Two theorem-proving based verifications of a similar design are described in [JSD98] and [PA98]. The idea in [JSD98] is to first show that for every out-of-order execution sequence that contains as many as n unretired instructions at any time there exists an "equivalent" (*max-1*) execution containing at most 1 unretired instruction by constructing a suitable controller schedule. It then shows the equivalence between a max-1 execution and the ISA level. The induction required in the first step, which was not mechanized, is very complicated. The verifier needs a much deeper insight into the control logic to exhibit a control schedule and to discharge the generated obligations in the first step than that is needed for constructing the completion functions and discharging the generated verification conditions. Whereas our verification makes no assumption on the time taken by the execution units, the mechanized part of their first step bounds the execution time. The proofs mix safety and liveness issues and the verification of liveness issues is not addressed. And the complexity of the reachability invariants needed in their approach and the effort required to discharge them is not clear; few details are provided in the paper.

The verification in [PA98] is based on *refinement* by using "synchronization on instruction retirement" to reduce the complexity of the refinement relations to be proved. Although they do not need any flushing mechanism, there is no systematic method to generate the invariants and obligations needed and hence their mechanization is not as automatic as ours. And they do not address liveness issues needed to complete the proof.

In [SH98], verification of a processor model with a reorder buffer, exceptions, and speculative execution is carried out. Their approach relies on constructing an explicit intermediate abstraction (called MAETT) and expressing invariant properties over this. Our approach avoids the construction of an intermediate abstraction and hence requires significantly less manual effort.

In [McM98], McMillan uses compositional model checking and aggressive symmetry reductions to manually decompose the proof of a processor implementing Tomasulo's algorithm (without a reorder buffer) into smaller correctness obligations via refinement maps. Setting up the refinement maps requires information similar to that provided by the completion functions in addition to some details of the design. An advantage of model checking is that it does not need any reachability invariants to check the refinement maps although the user has to give hints about the environment assumptions to be used.

The rest of the paper is organized as follows: In Section 2, we describe our processor model. Section 3 describes our correctness criteria and provides a brief overview of our approach applied to examples mentioned earlier in [HSG98]. This is followed by the proof of correctness in Section 4 and finally the conclusions.

2 Processor Model

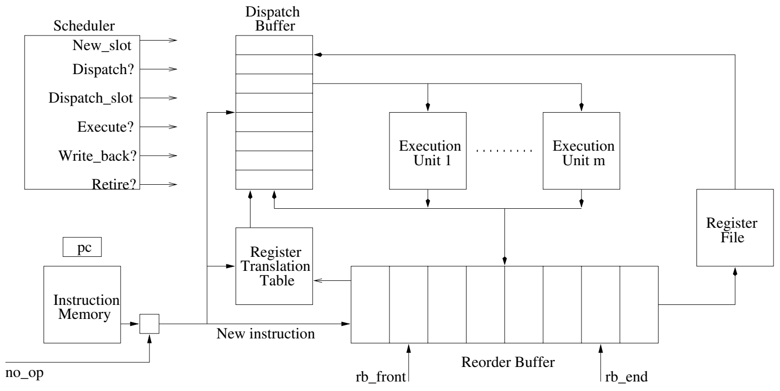

Fig. 1. The block diagram model of our implementation

The implementation model of an out-of-order execution processor that we consider in this paper is shown in Figure 1. A reorder buffer is used to maintain the program order of the instructions so that they can be committed in that order to respect the ISA semantics. (`rb_end` points to the earliest issued instruction and `rb_front` points to the first available free slot in the buffer). A register translation table (RTT) is maintained to provide the identity of the latest pending instruction writing a particular register. The model has a dispatch buffer (of size `z`; the dispatch buffer entries are also called "reservation stations" in other literature) where instructions wait before being sent to the execution units. There are `m` execution units represented by an uninterpreted function (`z` and `m` are parameters to our implementation model). A scheduler controls the movement of the instructions through the execution pipeline (such as being dispatched, executed etc.) and its behavior is modeled by axioms (to allow us to concentrate on the processor "core"). Instructions are fetched from the instruction memory (using a program counter which then is incremented); and the implementation also takes a `no_op` input, which suppresses an instruction fetch when asserted.

An instruction is *issued* by allocating an entry for it at the front of the reorder buffer and a free entry in the dispatch buffer (`New_slot`). No instruction is issued if the dispatch buffer is full or if `no_op` is asserted. The RTT entry corresponding to the destination of the instruction is updated to reflect the fact that the instruction being issued is the latest one to write that register. If the source operand is not being written by a previously issued pending instruction (checked using the RTT) then its value is obtained from the register file, otherwise the reorder buffer index of the instruction providing the source operand is maintained (in the dispatch buffer entry). Issued instructions wait in the dispatch buffer for their source operand to become ready, monitoring the execution units if they produce the value they are waiting for. An instruction can be *dispatched* when its source operand is ready and a free execution unit is available. `Dispatch?` and `Dispatch_slot` outputs from the scheduler (each a `m`-wide vector) determine whether or not to dispatch an instruction to a particular execution unit and the dispatch buffer entry from where to dispatch. As soon as an instruction is dispatched, its dispatch buffer entry is freed. Dispatched instructions get *executed* after a non-deterministic amount of time as determined by the scheduler output `Execute?`. The result of executed instructions are *written back* to their respective reorder buffer entries as well as forwarded to those instructions waiting for this result (at a time determined by the `Write_back?` output of the scheduler). If the instruction at the end of the reorder buffer has written back its result, then that instruction can be retired by copying the result value to the register file (at a time determined by the `Retire?` output of the scheduler). Also, if the RTT entry for the destination of the instruction being retired is pointing to the end, then that entry is updated to reflect the fact that value of that register is in the register file.

Our simplified model does not have memory or branch instructions and does not handle exceptions. For simplicity, multiple instruction issue or retirement is not allowed in a single cycle (but multiple instructions can be simultaneously dispatched or written back). Also, the reorder buffer is implemented as an unbounded buffer as opposed to a circular queue.[1]

At the specification level, the state is represented by a register file, a program counter and an instruction memory. Instructions are fetched from the instruction memory, executed, result written back to the register file and the program counter incremented in one clock cycle.

3 Our Correctness Criteria

Intuitively, a pipelined processor is correct if the behavior of the processor starting in a flushed state (i.e., no partially executed instructions), executing a program and terminating in a flushed state is emulated by an ISA level specification machine whose starting and terminating states are in direct correspondence through projection. This criterion is shown in Figure 2(a) where `I_step` is the implementation transition function, `A_step` is the specification transition function, and `projection` extracts those implementation state components visible to the specification (i.e., observables). This criterion can be proved by an easy induction on `n` once the *commutative diagram* condition shown in Figure 2(b)

[1] Using a bounded reorder buffer will not complicate the methodology but makes setting up the induction more involved.

is proved on a single implementation machine transition (and a certain other condition discussed in the next paragraph holds).

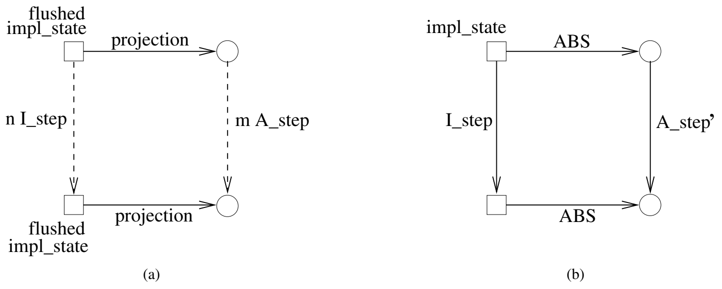

Fig. 2. Pipelined microprocessor correctness criteria

The criterion in Figure 2(b) states that if the implementation machine starts in an arbitrary reachable state `impl_state` and the specification machine starts in a corresponding specification state (given by an abstraction function `ABS`), then after executing a transition their new states correspond. Further `ABS` must be chosen so that for all flushed states `fs` the *projection condition* `ABS(fs) = projection(fs)` holds. The commutative diagram uses a modified transition function `A_step'`, which denotes zero or more applications of `A_step`, because an implementation transition from an arbitrary state might correspond to executing in the specification machine zero instruction (*e.g.*, if the implementation machine stalls without fetching an instruction) or more than one instruction (*e.g.*, if multiple instructions are fetched in a cycle). The number of instructions executed by the specification machine is provided by a user-defined *synchronization* function on implementation states. One of the crucial proof obligations is to show that this function does not always return zero (*No_indefinite_stutter* obligation). One also needs to prove that the implementation machine will eventually reach a flushed state if no more instructions are inserted into the machine, to make sure that the correctness criterion in Figure 2(a) is not vacuous (*Eventual_flush* obligation). In addition, the user may need to discover *invariants* to restrict the set of `impl_state` considered in the proof of Figure 2(b) and prove that it is closed under `I_step`.

The completion functions approach suggests a way of constructing the abstraction function. We define a completion function for every unfinished instruction in the processor that directly specifies the intended effect of completing that instruction. The abstraction function is defined as a composition of these completion functions in program order. In the examples in [HSG98], the program order was determined from the structure of the pipeline. This construction of the abstraction function decomposed the proof into proving a series of verification conditions, each of which captured the effect of completing instructions one at a time and that were reused in the proof of the subsequent verification conditions. Since there were a fixed (and small) number of instructions pending in the pipeline, this scheme worked well and the proof was easily accomplished.

However, the number of instructions is unbounded in the present example and the above scheme does not work. But we observe that a pending instruction in the processor can only be in four possible states and provide a parameterized completion function using this fact. The program order is easily determined since the reorder buffer stores it. And we generate a single parameterized verification condition which is proved by an induction on the number of pending instructions in the reorder buffer, where the induction hypothesis captures the effect of completing all the earlier instructions.

4 Proof of Correctness

We introduce some notations which will be used throughout this section: `q` represents the implementation state, `s` the scheduler output, `i` the processor input, `rf(q)` the register file contents in state `q` and `I_step(q,s,i)` the "next state" after an implementation transition. Also, we identify an instruction in the processor by its reorder buffer entry index (i.e., instruction `rbi` means instruction at index `rbi`). The complete PVS specifications and the proof scripts can be found at [Hos99].

4.1 Specifying the completion functions

An instruction in the processor can be in one of the following four possible states inside the processor—issued, dispatched, executed or written back. (A retired instruction is no longer present in the processor). We formulate predicates describing an instruction in each of these states and identify how to complete such an instruction. To facilitate this formulation, we add two auxiliary variables to a reorder buffer entry.[2] The first one maintains the index of the dispatch buffer entry allocated to the instruction while it is waiting to be dispatched. The second one maintains the execution unit index where the instruction executes. The definition of the completion function is shown in $\boxed{1}$.

```
% state_I:impl. state type.  rbindex:reorder buffer index type.       1
Complete_instr(q:state_I,rbi:rbindex):state_I =
   IF written_back_predicate(q,rbi) THEN Action_written_back(q,rbi)
   ELSIF executed_predicate(q,rbi) THEN Action_executed(q,rbi)
   ELSIF dispatched_predicate(q,rbi) THEN Action_dispatched(q,rbi)
   ELSIF issued_predicate(q,rbi) THEN Action_issued(q,rbi)
   ELSE q ENDIF
```

In this implementation, when the instruction is in the written back state, the result value as well as the destination register of the instruction are in its reorder buffer entry. So `Action_written_back` above completes this instruction by updating the register file by writing the result value to the destination register. An instruction in the issued state is completed (`Action_issued`) by reading the

[2] The auxiliary variables are for specification purposes only. The third auxiliary variable we needed maintained the identity of the source register for a given instruction.

value of the source register from the register file, (this relies on the fact that the completion functions will be composed in the program order in defining the abstraction function; so `q` for a given instruction will be that state where the instructions ahead of it are completed) computing the result value depending on the instruction operation and then writing this value to the destination register. Similarly `Action_executed` and `Action_dispatched` are specified. None of these "actions" affect the program counter or the instruction memory. The completion function definition is very compact, taking only 15 lines of PVS code.

4.2 Constructing the abstraction function

The abstraction function is constructed by flushing the reorder buffer, that is, by completing all the unfinished instructions in the reorder buffer. We define a recursive function `Complete_till` to complete instructions till a given reorder buffer index as shown in $\boxed{2}$ and then construct the abstraction function by instantiating this definition with the index of the latest instruction in the reorder buffer (i.e., `rb_front(q)-1`). The synchronization function returns zero if `no_op` input is asserted or there is no free dispatch buffer entry (hence no instruction is issued) otherwise returns one.

```
% If the given instr. index is less than the end pointer of the        2
% reorder buffer, do nothing. Else complete that instr. in a state
% where all the previous instructions are completed.
Complete_till(q:state_I,rbi:rbindex): RECURSIVE state_I =
   IF rbi < rb_end(q) THEN q
   ELSE Complete_instr(Complete_till(q,rbi-1),rbi) ENDIF
   MEASURE rbi
% state_A is the specification state type.
ABS(q:state_I):state_A = projection(Complete_till(q,rb_front(q)-1))
```

4.3 Proof decomposition

We first prove a single parameterized verification condition that captures the effect of completing all the instructions in the reorder buffer and then use it in the proof of the commutative diagram. We decompose the proof of this verification condition based on how an instruction makes a transition from its present state to its next state.

Consider an arbitrary instruction `rbi`. We claim that the register file contents will be the same whether the instructions till `rbi` are completed in state `q` or in `I_step(q,s,i)`. This is shown as lemma `same_rf` in $\boxed{3}$. We prove this by induction on `rbi`.

```
% The single parametrized verification condition.                      3
% valid_rb_entry? predicate tests if rbi is within reorder buffer bounds.
same_rf: LEMMA
    FORALL(rbi:rbindex): valid_rb_entry?(q,rbi) IMPLIES
    rf(Complete_till(q,rbi)) = rf(Complete_till(I_step(q,s,i),rbi))
```

We generate the different cases of the induction argument (as detailed later) based on how an instruction makes a transition from its present state to its next state. This is shown in Figure 3 where we have identified the conditions under which an instruction changes its state. For example, we identify the predicate `Dispatch_trans?(q,s,i,rbi)` that defines the condition under which the instruction `rbi` goes from issued state to dispatched state. In this implementation, this predicate is true when there is an execution unit for which `Dispatch?` output from the scheduler is true and the `Dispatch_slot` output is equal to the dispatch buffer entry index assigned to `rbi`. Similarly other "trans" predicates are defined.

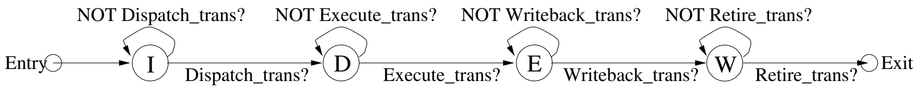

Fig. 3. The various states an instruction can be in and transitions between them, I: issued, D: dispatched, E: executed, W: written back.

Having defined these predicates, we prove that they indeed cause instructions to take the transitions shown. Consider a valid instruction `rbi` in the issued state, that is, `issued_predicate(q,rbi)` holds. If `Dispatch_trans?(q,s,i,rbi)` is true, then we show that after an implementation transition, `rbi` will be in the dispatched state (i.e., `dispatched_predicate(I_step(q,s,i),rbi)` is true) and remains valid. This is shown as a lemma in $\boxed{4}$. If `Dispatch_trans?(q,s,i,rbi)` is false, we show that `rbi` remains in the issued state in `I_step(q,s,i)` and remains valid. There are five other similar lemmas for the other transitions. In the eighth case, that is, `rbi` in the written back state being retired, the instruction will be invalid (out of reorder buffer bounds) in `I_step(q,s,i)`.

```
issue_to_dispatch: LEMMA                                              4
    FORALL(rbi:rbindex): (valid_rb_entry?(q,rbi) AND
     issued_predicate(q,rbi) AND Dispatch_trans?(q,s,i,rbi)) IMPLIES
    (dispatched_predicate(I_step(q,s,i),rbi) AND
       valid_rb_entry?(I_step(q,s,i),rbi))
```

Now we come back to details of the induction argument for `same_rf` lemma. We do a case analysis on the possible state `rbi` is in and whether or not, it makes a transition to its next state. Assume the instruction `rbi` is in the issued state. We prove the induction claim in the two cases—`Dispatch_trans?(q,s,i,rbi)` is true or false—separately. (The proof obligation for the first case is shown in $\boxed{5}$.) We have similar proof obligations for `rbi` being in other states. In all, the proof decomposes into eight very similar proof obligations.

```
% One of the eight cases in the induction argument.                    5
issue_to_dispatch_induction: LEMMA
    FORALL(rbi:rbindex): (valid_rb_entry?(q,rbi) AND
     issued_predicate(q,rbi) AND Dispatch_trans?(q,s,i,rbi) AND
     Induction_hypothesis(q,s,i,rbi-1)) IMPLIES
    rf(Complete_till(q,rbi)) = rf(Complete_till(I_step(q,s,i),rbi))
```

We now sketch the proof of `issue_to_dispatch_induction` lemma. (We refer to the goal that we are proving—`rf(...) = rf(...)`—as the consequent.) We expand the definition of the completion function corresponding to `rbi` on both sides of the consequent (after unrolling the recursive definition of `Complete_till` once). It follows from `issue_to_dispatch` lemma that since `rbi` is in issued state in `q`, it is in dispatched state in `I_step(q,s,i)`. After rewriting and simplifications in PVS, the left hand side of the consequent simplifies to `rf(Action_issued(Complete_till(q,rbi-1),rbi))` [3] and the right hand side to `rf(Action_dispatched(Complete_till(I_step(q,s,i),rbi-1),rbi))` (Illustrated in Figure 4). Proof now proceeds by expanding the definitions of `Action_issued` and `Action_dispatched`, using the necessary invariants and simplifying. We use many simple PVS strategies during the proof; in particular we use `(apply (then* (repeat (lift-if)) (bddsimp) (ground) (assert)))` to do the simplifications by automatic case-analysis. Observe that when we expand `Action_dispatched`, all implementation variables take their "next" values. Also on the left hand side of the consequent, term `rf(Complete_till(q,rbi-1))` appears and on right hand side, term `rf(Complete_till(I_step(q,s,i),rbi-1))` appears and these are same by the induction hypothesis.

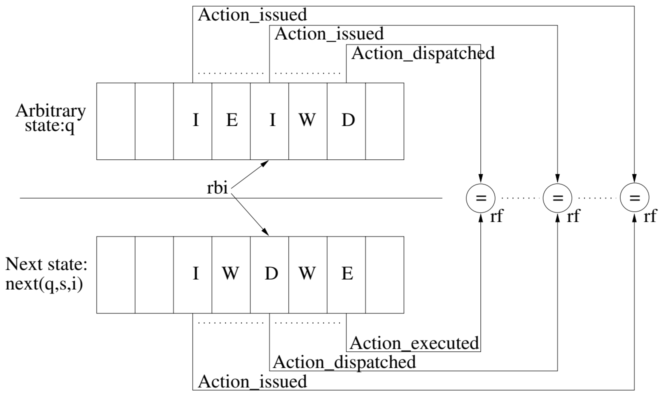

Fig. 4. The reorder buffer and the state of the instructions in it before and after an implementation transition (one possible configuration, empty slot means no instruction present). Completing a particular instruction reduces to performing the action shown.

We now instantiate the lemma `same_rf` above with the index of the latest instruction in the processor (i.e., `rb_front(q)-1`) and use it in the proof of the commutative diagram for register file. Assume that no instruction is issued in the current cycle, that is, the synchronization function returns zero. Then `rb_front` remains unchanged after an implementation transition and the proof is trivial. If indeed a new instruction is issued, then it will be at index `rb_front(q)` and will be in issued state, so proving the commutative diagram reduces to establishing that completing the new instruction (as per `Action_issued`) has the same effect

[3] Observe that `issued_predicate(Complete_till(q,rbi-1),rbi)` if and only if `issued_predicate(q,rbi)`. This is because the completion functions affect only the register file (observables in general) and `issued_predicate` depends only on the non-observables.

on the register file as executing a specification machine transition. This proof is similar to the proof of the lemma described above. The commutative diagram proofs for `pc` and the instruction memory are trivial and are omitted.

Correctness of feedback logic: The proof presented above requires that the correctness of the feedback logic be captured in the form of a lemma as shown in $\boxed{6}$. This lemma states that if the source operand of an instruction is ready, then its value is equal to the value read from the register file after all the instructions ahead of it are completed. When an instruction `rbi` in the issued state is being dispatched, it uses `src_value(q,rbi)` as the source operand but the `Action_issued` that is used to complete it reads the source value from the register file (see the description of `Action_issued` in Section 4.1) and this lemma establishes that these two values are the same. The proof of this lemma relies on an invariant described later.

```
% select reads from the register file. src_ready?, src_value and      6
% src_reg have their obvious definitions.
Feedback_logic_correct: LEMMA
    FORALL(rbi:rbindex): (valid_rb_entry?(q,rbi) AND
     issued_predicate(q,rbi) AND src_ready?(q,rbi)) IMPLIES
    src_value(q,rbi) = select(rf(Complete_till(q,rbi-1)),src_reg(q,rbi))
```

Invariants needed: We now provide a classification of the invariants needed by our approach and describe some of them.

- *Exhaustiveness and Exclusiveness*: Having identified the set of possible states an instruction can be in, we require one to prove that an arbitrary instruction is always in one of these states (exhaustiveness) and never simultaneously in two states (exclusiveness).
- *Instruction state properties*: Whenever an instruction is in a particular state, it satisfies some properties and these are established as invariants. One example is: if an instruction is in issued state, then the dispatch buffer entry assigned to it is valid and has the reorder buffer index of the instruction stored in it.
- *Feedback logic invariant*: Let `rbi` be an arbitrary instruction and let `ptr` be an instruction that is producing its source value. Then this invariant essentially states that all the instructions "in between" `rbi` and `ptr` have the destination different from the source of `rbi`, that `ptr` is in the written back state if and only if the source value of `rbi` is ready and that the source value of `rbi` (when ready) is equal to the result computed by `ptr`.
- *Example specific invariants*: Other invariants needed include characterization about the reorder buffer bounds and the register translation table.

PVS proof details: The proofs of all the induction obligations follow the pattern outlined in the sketch of `issue_to_dispatch_induction` lemma. The proofs of certain rewrite rules needed in the methodology [HSG98] and other simple obligations can be accomplished fully automatically. But there is no uniform strategy for proving the invariants. The manual effort involved one week of discussion and planning and then 12 person days of "first time" effort to construct the proofs. The proofs got subsequently cleaned up and evolved as we wrote the paper. The proofs rerun in about 1050 seconds on a 167 MHz Ultra Sparc machine.

4.4 Other obligations - liveness properties

We provide a sketch of the proof that the processor eventually gets flushed if no more instructions are inserted into it. The proof that the synchronization function eventually returns a nonzero value is similar. The proofs involve a set of obligations on the implementation machine, a set of fairness assumptions on the inputs to the implementation and a high level argument using these to prove the two liveness properties. All the obligations on the implementation machine are proved in PVS. We now provide a brief sketch (due to space constraints) of the top level argument which is being formalized in PVS.
Proof sketch: The processor is flushed if `rb_front(q) = rb_end(q)`.

- First observation: "any instruction in the dispatched state eventually goes to the executed state and then eventually goes to the written back state. It then remains in the written back state until retired". Consider an instruction `rbi` in the dispatched state. If `Execute_trans?(q,s,i,rbi)` is true, then `rbi` goes to the executed state in `I_step(q,s,i)`, otherwise it remains in the dispatched state (refer to Figure 3). We show that when `rbi` is in the dispatched state, the scheduler inputs that determine when an instruction should be executed are enabled and these remain enabled as long as `rbi` is in the dispatched state. By a fairness assumption on the scheduler, it eventually decides to execute the instruction (i.e., `Execute_trans?(q,s,i,rbi)` will be true) and the instruction moves to the executed state. By a similar argument, it moves to the written back state and then remains in that state until retired.
- Second observation: "every busy execution unit eventually becomes free and stays free until an instruction is dispatched on it".
- Third observation: "an instruction in the issued state will eventually go to the dispatched state". Here, the proof is by induction since an arbitrary instruction `rbi` could be waiting for a previously issued instruction to produce its source value. This step also relies on the earlier two observations.
- Final observation: "the processor eventually gets flushed". We know that every instruction eventually goes to the written back state—third and first observations. Also the instructions in the written back state are eventually retired by a fairness assumption on the scheduler. Since `rb_front(q)` remains unchanged when no new instructions are inserted into the processor and `rb_end(q)` is incremented when an instruction is retired, eventually the processor gets flushed.

5 Conclusions

We have demonstrated in this paper that the completion functions approach is well-suited for the verification of out-of-order execution processors with a reorder buffer. We have recently extended our approach to be applicable in a scenario where instructions "commit" out-of-order and illustrated it on an example processor implementing Tomasulo's algorithm without a reorder buffer [HGS99]. The proof was constructed in seven person days, reusing lot of the ideas and the machinery developed in this paper. We are currently working on verifying a more detailed out-of-order execution processor involving branches, exceptions and speculative execution. Our approach has been used to handle processors with branch and memory operations [HSG98] and we are investigating how those ideas carry over to this example. We are also developing a PVS theory of the "eventually" temporal operator to mechanize the liveness proofs presented in this paper. Finally, we are investigating how the ideas behind the completion functions approach can be adapted to verify certain "transaction processing systems".

Acknowledgments: We would like to thank Abdel Mokkedem and John Rushby for their feedbacks on the earlier drafts of this paper.

References

BDL96. Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design, FMCAD '96*, volume 1166 of *LNCS*, pages 187–201. Springer-Verlag, November 1996. 47

CRSS94. D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design, TPCD '94*, volume 910 of *LNCS*, pages 203–222. Springer-Verlag, September 1994. 47

GW98. Ganesh Gopalakrishnan and Phillip Windley, editors. *Formal Methods in Computer-Aided Design, FMCAD '98*, volume 1522 of *LNCS*, Palo Alto, CA, USA, November 1998. Springer-Verlag. 59, 59

HGS99. Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas. A proof of correctness of a processor implementing Tomasulo's algorithm without a reorder buffer. 1999. Submitted for publication. 58

Hos99. Ravi Hosabettu. PVS specification and proofs of all the examples verified with the completion functions approach, 1999. Available at `http://www.cs.utah.edu/~hosabett/pvs/processor.html`. 53

HP90. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990. 48

HSG98. Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In Hu and Vardi [HV98], pages 122–134. 47, 48, 48, 50, 52, 57, 59

HV98. Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *LNCS*, Vancouver, BC, Canada, June/July 1998. Springer-Verlag. 59, 59, 59

JSD98. Robert Jones, Jens Skakkebaek, and David Dill. Reducing manual abstraction in formal verification of out-of-order execution. In Gopalakrishnan and Windley [GW98], pages 2–17. 49, 49

McM98. Ken McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In Hu and Vardi [HV98], pages 110–121. 49

ORSvH95. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995. 48

PA98. Amir Pnueli and Tamarah Arons. Verification of data-insensitive circuits: An in-order-retirement case study. In Gopalakrishnan and Windley [GW98], pages 351–368. 49, 49

SH98. J. Sawada and W. A. Hunt, Jr. Processor verification with precise exceptions and speculative execution. In Hu and Vardi [HV98], pages 135–146. 49