# Verifying Advanced Microarchitectures that Support Speculation and Exceptions *

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

[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.** In this paper, we discuss the verification of a microprocessor involving a reorder buffer, a store buffer, speculative execution and exceptions at the microarchitectural level. We extend the earlier proposed *Completion Functions Approach* [HSG98] in a uniform manner to handle the verification of such microarchitectures. The key extension to our previous work was in systematically extending the abstraction map to accommodate the possibility of all the pending instructions being squashed. An interesting detail that arises in doing so is how the commutativity obligation for the program counter is proved despite the program counter being updated by both the instruction fetch stage (when a speculative branch may be entertained) and the retirement stage (when the speculation may be discovered to be incorrect). Another interesting detail pertains to how store buffers are handled. We highlight a new type of invariant in this work—one which keeps correspondence between store buffer pointers and reorder buffer pointers. All these results, taken together with the features handled using the completion functions approach in our earlier published work [HSG98,HSG99,HGS99], demonstrates that the approach is uniformly applicable to a wide variety of pipelined designs.

## 1 Introduction

Formal Verification of pipelined processor implementations against instruction set architecture (ISA) specifications is a problem of growing importance. A significant number of processors being sold today employ advanced features such as out-of-order execution, store buffers, exceptions that cause pending uncommitted instructions to be squashed, and speculative execution. Recently a number of different approaches [HSG99,McM98,PA98] have been used to verify simple out-of-order designs. To the best of our knowledge, no single formal verification

* The first and second authors were supported in part by NSF Grant No. CCR-9800928. The third author was supported in part by ARPA contract F30602-96-C-0204 and NASA contract NASI-20334. The first author was also supported by a University of Utah Graduate Fellowship.

E.A. Emerson and A.P. Sistla (Eds.): CAV 2000, LNCS 1855, pp. 521–537, 2000.

technique has been shown to be capable of verifying processor designs that support all of these features and also apply to other processors such as those that perform out-of-order retirement. In this paper, we report our successful application of the *Completion Functions Approach* to verify an out-of-order execution design with a reorder buffer, a store buffer, exceptions and speculation, using the PVS [ORSvH95] theorem-prover, taking only a modest amount of time for the overall proof. This result, taken together with the earlier published applications of the completion functions approach [HSG98,HSG99,HGS99], demonstrates that the approach is uniformly applicable to a wide variety of pipelined designs.

One of the challenges posed in verifying a combination of the above mentioned advanced features is that the resulting complex interaction between data and control usually overwhelms most automatic methods, whether based on model checking or decision procedures. One of the main contributions of this work is that we develop a way of cleanly decomposing the squashing of instructions from normal execution. These decomposition ideas are applicable to theorem proving or model checking or combined methods.

Our basic approach is one of showing that any program run on the specification and the implementation machines returns identical results. This verification is, in turn, achieved by identifying an abstraction map `ABS` that relates implementation states to corresponding specification states. The key to make the above technique work efficiently in practice is a proper definition of `ABS`. As we showed, in our earlier work [HSG98], one should ideally choose an approach to constructing `ABS` that is not only simple and natural to carry out, but also derives other advantages, the main ones being *modular verification* that helps localize errors, and *verification reuse* that allows lemmas proved about certain pipeline stages to be used as rewrite rules in proving other stages. In [HSG98], we introduced such a technique to define `ABS` called the *Completion Functions Approach*. In subsequent work [HSG99,HGS99,Hos99], we demonstrated that the completion functions approach can be applied uniformly to a wide variety of examples that include various advanced pipelining features. An open question in our previous work was whether combining out-of-order execution with exceptions and speculation would make the task of defining completion functions cumbersome and the approach impractical.

In this paper, we demonstrate that the completion functions approach is robust enough to be used effectively for such processors, that is, (i) the specification of completion functions are still natural, amounting to expressing knowledge that the designer already has; (ii) verification proceeds incrementally, facilitating debugging and error localization; (iii) mistakes made in specifying completion functions never lead to false positives; and (iv) verification conditions and most of the supporting lemmas needed to finish a proof can be generated systematically, if not automatically. They can also be discharged with a high degree of automation using strategies based on decision procedures and rewriting. These observations are supported by our final result: a processor design supporting superscalar execution, store buffers, exceptions, speculative branch prediction, and user and supervisor modes could be fully verified in 265 person hours. This,

we believe, is a modest investment in return for the significant benefits of design verification.

Some of the highlights of the work we report are as follows. Given that our correctness criterion is one of showing a commutativity obligation between implementation states and specification states, the abstraction map used in the process must somehow accommodate the possibility of instructions being squashed. We show how this is accomplished. This leads us to a verification condition with two parts, one pertaining to the processor states being related before and after an implementation transition, and the other relating to the squashing predicate itself. Next, we show how the commutativity obligation for the program counter is obtained despite the program counter being updated by both the instruction fetch stage (when a speculative branch may be entertained) and the retirement stage (when the speculation may be discovered to be incorrect). We also show how the store buffer is handled in our proof. We detail a new type of invariant in this work, which was not needed in our earlier works. This invariant keeps correspondence between store buffer pointers and reorder buffer pointers.

## 2 Processor Model

At the specification level, the state of the processor is represented by a register file, a special register file accessed only by privileged/special instructions, a data memory, a mode flag, a program counter and an instruction memory. The processor operating mode (one of user/supervisory) is maintained in the mode flag. User mode instructions are an `alu` instruction for performing arithmetic and logical operations, `load` and `store` instructions for accessing the data memory, and a `beq` instruction for performing conditional branches. Three additional privileged instructions are allowed in the supervisory mode: `rfeh` instruction for returning from an exception handler, and `mfsr` and `mtsr` instructions for moving data from and to the special register file. Three types of exceptions are possible: arithmetic exception raised by an `alu` instruction, data access exception raised by `load` and `store` instructions when the memory address is outside legal bounds (two special registers maintain the legal bounds, and this is checked only in user mode), and an illegal instruction exception. When an exception is raised, the processor saves the address of the faulting instruction in a special register and jumps to an exception handler assuming supervisory mode in the process. After processing a raised exception, the processor returns to user mode via the `rfeh` instruction.

An implementation model of this processor is shown in Figure 1. A reorder buffer, implemented as a circular FIFO queue with its tail pointing to the earliest issued instruction and head pointing to the first free location in the buffer, is used to maintain program order, to permit instructions to be committed in that order. Register translation tables (regular and special) provide the identity of the latest pending instruction writing a particular register. "Alu/Branch/Special Instr. Unit" (referred to as ABS Unit) executes `alu`, `beq` and all the special instructions. The reservation stations hold the instructions sent to this unit

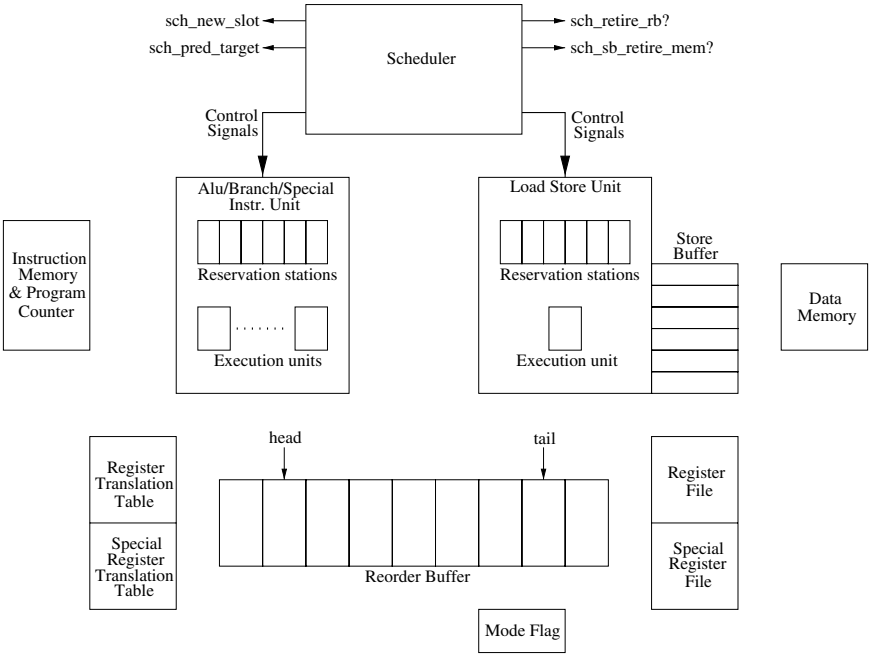


**Fig. 1.** The block diagram model of our implementation

until they are ready to be dispatched onto an appropriate execution unit. These instructions are executed out of program order by the multiple execution units present in the ABS Unit. Instructions `load` and `store` are issued to the "Load Store Unit" (referred to as LS Unit) where the reservation stations form a circular FIFO queue storing the instructions in their program order. (Again, tail points to the earliest instruction and head points to the first free reservation station.) These instructions are executed in their program order by the single execution unit present in the LS Unit. For a `store` instruction, the memory address and the value to be stored are recorded in an entry in the *store buffer*, and the value is later written into the data memory. The store buffer is again implemented as a circular FIFO queue, with head and tail pointers, keeping the instructions to be written to the data memory in their program order. When two store buffer entries refer to the same memory address, the latest one has a flag set. A `load` instruction first attempts an associative search in the store buffer using the memory address. If multiple store buffer entries have the same address, the search returns the value of the latest entry. If the search does not find a matching entry, the data for that address is returned from the data memory. 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; 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 head of the reorder buffer and (depending on the instruction type) either a free reservation station (`sch_new_slot`) in the ABS Unit or a free reservation station at the head of the queue of reservation stations in the LS Unit. If the instruction being issued is a branch instruction, then the program counter is modified according to a *predicted* branch target address (`sch_pred_target`, an unconstrained arbitrary value), and in the next cycle the new instruction is fetched from this address. No instruction is issued if there are no free reservation stations/reorder buffer entries or if `no_op` is asserted or if the processor is being restarted (for reasons detailed later). 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 operands are not being written by previously issued pending instructions (checked using the RTT) then their values are obtained from the register file, otherwise the reorder buffer indices of the instructions providing the source operands are maintained (in the reservation station). Issued instructions wait for their source operands to become ready, monitoring all the execution units if they produce the values they are waiting for. An instruction can be *dispatched* when its source operands are ready and a free execution unit is available [1]. In case of the LS Unit, only the instruction at the tail of the queue of reservation stations is dispatched. As soon as an instruction is dispatched, its reservation station is freed. The dispatched instructions are *executed* and the results are *written back* to their respective reorder buffer entries as well as forwarded to those instructions waiting for this result. If an exception is raised by any of the executing instructions, then a flag is set in the reorder buffer entry to indicate that fact. In case of a `store` instruction, the memory address and the value to be stored are written into a store buffer entry instead of the reorder buffer entry when the `store` instruction does not raise an exception (other information such as the "ready" status etc. are all written into the reorder buffer entry). The control signals from the scheduler determine the timings of this movement of the instructions in the execution pipeline.

The instruction at the tail of the reorder buffer is committed to the architecturally visible components, when it is done executing (at a time determined by `sch_retire_rb?`). If it is a `store` instruction, then the corresponding store buffer entry is marked *committed* and later written into the data memory (at a time determined by `sch_sb_retire_mem?`). Also, if the RTT entry for the destination of the instruction being retired is pointing to the tail of the reorder buffer, then that RTT entry is updated to reflect the fact that the value of that register is in the appropriate register file. If the instruction at the tail of the reorder buffer has raised an exception or if it is a mis-predicted branch or if it is a `rfeh` in-

[1] Multiple instructions can be simultaneously dispatched, executed and written back in one clock cycle. However, for simplicity, we do not allow multiple instruction issue or retirement in a single clock cycle.

struction, then the rest of the instructions in the reorder buffer are squashed and the processor is restarted by resetting all of its internal (non-observable) state.

## 3 The Completion Functions Approach

The key idea in proving the correctness of pipelined microprocessors is to discover a formal correspondence between the execution of the implementation and the specification machines. The completion functions approach suggests a way of constructing this abstraction in a manner that leads to an elegant decomposition of the proof. In the first subsection, we briefly discuss the correctness criterion we use. In the second subsection, we describe the different steps in constructing a suitable abstraction function for the example under consideration. In the third subsection, we discuss how to decompose the proof into verification conditions, the proof strategies used in discharging these obligations, and the invariants needed in our approach. The PVS specifications and the proofs can be found at [Hos99].

### 3.1 Correctness Criterion

We assume that the pipelined implementation and the ISA-level specification are provided in the form of transition functions, denoted by `I_step` and `A_step` respectively. The specification machine state is made up of certain components chosen from the implementation machine called the observables. The function `projection` extracts these observables given an implementation machine state. The state where the pipelined machine has no partially executed instructions is called a *flushed* state.

We regard a pipelined processor implementation to be *correct* if the behavior of the processor starting in a flushed state, executing a program, and terminating in a flushed state is matched by the ISA level specification machine whose starting and terminating states are in direct correspondence with those of the implementation processor through `projection`. This criterion is shown in Figure 2(a) where `n` is the number of implementation machine transitions in a run of the pipelined machine and `m` corresponds to the number of instructions executed in the specification machine by this run. An additional correctness criterion is to show that the implementation machine is able to execute programs of all lengths, that is, it does not get into a state where it refuses to accept any more new instructions. In this paper, we concentrate on proving the correctness criterion expressed in Figure 2(a) only.

The criterion shown in Figure 2(a) spanning an entire sequential execution can be established with the help of induction once a more basic *commutativity obligation* shown in Figure 2(b) is established on a single implementation machine transition. This criterion states that if the implementation machine starts in an arbitrary state `q` 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. `A_step_new` stands for zero or more

applications of `A_step`. The number of instructions executed by the specification machine corresponding to an implementation transition is given by a user defined *synchronization* function. Our method further verifies that the `ABS` function chosen corresponds to `projection` on flushed states, that is, `ABS(fs) = projection(fs)` holds on flushed states, thus helping debug `ABS`. The user may also need to discover invariants to restrict the set of implementation states considered in the proof of the commutativity obligation and prove that it is closed under `I_step`.

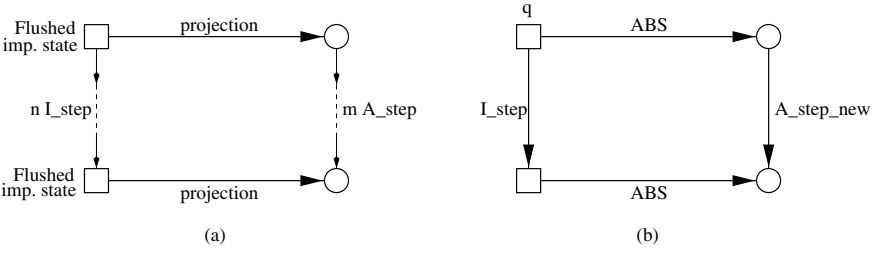


**Fig. 2.** Pipelined microprocessor correctness criterion

The crux of the problem here is to define a suitable abstraction function relating an implementation state to a specification state. The completion functions approach suggests a way of doing this in a manner that leads to an elegant decomposition of the proof. We now detail how this is achieved for our example processor.

### 3.2 Compositional Construction of the Abstraction Function

The first step in defining the abstraction function is to identify all the unfinished instructions in the processor and their program order. In this implementation, the processor (when working correctly) stores all the currently executing instructions in their program order in the reorder buffer. We identify an instruction in the processor with its reorder buffer index, that is, we refer to instruction at reorder buffer index `rbi` as just instruction `rbi`[2]. In addition to these, the store buffer has certain committed `store` instructions yet to be written into the data memory, recorded in their program order. These `store` instructions are not associated with any reorder buffer entry and occur earlier in the program order than all the instructions in the reorder buffer.

[2] Brief explanation of some of the notation used throughout rest of the paper: `q` refers to an arbitrary implementation state, `s` the scheduler output, `i` the processor input, `I_step(q,s,i)` the next state after an implementation transition. We sometimes refer to predicates and functions defined without explicitly mentioning their arguments, when this causes no confusion.

Having determined the program order of the unfinished instructions, the second step is to define a completion function for every unfinished instruction in the pipeline. Each completion function specifies the *desired effect* on the observables of completing a particular unfinished instruction assuming those that are ahead of it (in the program order) are completed. The completion functions, which map an implementation state to an implementation state, leave all non-observable state components unchanged. However not every instruction in the pipeline gets executed completely and updates the observables. If an instruction raises an exception or if the target address is mis-predicted for a branch instruction, then the instructions following it must be squashed. To specify this behavior, we define a squashing predicate for every unfinished instruction that is true exactly when the unfinished instruction can cause the subsequent instructions (in the program order) to be squashed. The completion function for a given instruction updates the observables only if the instruction is not squashed by any of the instructions preceding it.

We now elaborate on specifying the completion functions and the squashing predicates for the example under consideration. An unfinished instruction `rbi` in the processor can be in one of the following seven phases of execution: Issued to ABS Unit or to LS Unit (*issued_abs* or *issued_lsu*), dispatched in either of these units (*dispatched_abs* or *dispatched_lsu*), executed in either of these units (*executed_abs* or *executed_lsu*) or written back to the reorder buffer (*writtenback*). A given unfinished instruction is in one of these phases at any given time and the information about this instruction (the source values, destination register etc) is held in the various implementation components. For each instruction phase "ph", we define a predicate "Instr_ph?" that is true when a given instruction is in phase "ph", a function "Action_ph" that specifies what ought to be the effect of completing an instruction in that phase, and a predicate "Squash_rest?_ph" that specifies the conditions under which an instruction in that phase can squash all the subsequent instructions. We then define a single parameterized completion function and squashing predicate (applicable to all the unfinished instructions in the reorder buffer) as shown in 1. We similarly define (a parameterized) completion function for the committed `store` instructions in the store buffer. These `store` instructions can only be in a single phase, that is, *committed*, and they do not cause the subsequent instructions to be squashed. (A `store` instruction that raises an exception is not entered into the store buffer.)

1

```
% state_I: impl. state type. rbindex: reorder buffer index type.
Complete_instr(q:state_I, rbi:rbindex, kill?:bool): state_I =
  IF kill? THEN q
  ELSIF Instr_writtenback?(q,rbi) THEN Action_writtenback(q,rbi)
  ELSIF Instr_executed_lsu?(q,rbi) THEN Action_executed_lsu(q,rbi)
  ELSIF ... Similarly for other phases ... ENDIF

Squash_rest?_instr(q:state_I, rbi:rbindex): bool =
  IF Instr_writtenback?(q,rbi) THEN Squash_rest?_writtenback(q,rbi)
  ELSIF Instr_executed_lsu?(q,rbi) THEN Squash_rest?_executed_lsu(q,rbi)
  ELSIF ... Similarly for other phases ... ENDIF
```

In this implementation, when an instruction is in the *writtenback* phase, its reorder buffer entry has the result value and destination of the instruction, and also enough information to determine whether it has raised any exceptions or has turned out to be a mis-predicted branch. `Action_writtenback` and `Squash_rest?_writtenback` are then defined using this information about the instruction. Similarly, we define the "Action"s and the "Squash_rest?"s for the other phases. When an instruction is in an execution phase where it has not yet read its operands, the completion function obtains the operands by simply reading them from the observables. The justification is that the completion functions are composed in their program order in constructing the abstraction function (described below), and so we talk of completing a given instruction in a context where the instructions ahead of it are completed.

2

```
% Complete_Squash_rest?_till returns a tuple.
% proj_1 and proj_2 extracts the first and the second components.
% rbindex_p is type ranging from 0 to the size of the reorder buffer.
Complete_Squash_rest?_till(q:state_I,rbi_ms:rbindex_p):
                                          RECURSIVE [state_I,bool] =
 IF rbi_ms = 0 THEN (q,FALSE)
 ELSE LET t = Complete_Squash_rest?_till(q,rbi_ms-1),
          x = proj_1(t), y = proj_2(t) IN
    (Complete_instr(x,measure_fn_rbi(q,rbi_ms),y), %% 1st component.
     Squash_rest?_instr(x,measure_fn_rbi(q,rbi_ms)) OR y) %% 2nd one.
 ENDIF
 MEASURE rbi_ms

Complete_till(q:state_I,rbi_ms:rbindex_p): state_I =
 proj_1(Complete_Squash_rest?_till(Complete_committed_in_sb_till(
                                   q,lsu_sb_commit_count(q)),rbi_ms))

Squash_rest?_till(q:state_I,rbi_ms:rbindex_p): bool =
 proj_2(Complete_Squash_rest?_till(Complete_committed_in_sb_till(
                                   q,lsu_sb_commit_count(q)),rbi_ms))

% state_A is the specification state type.
ABS(q: state_I): state_A = projection(Complete_till(q,rb_count(q)))
```

The final step is to construct the abstraction function (that has the cumulative effect of flushing the pipeline) by completing all the unfinished instructions in their program order. A given instruction is to be killed, that is, the `kill?` argument of `Complete_instr` is true, when the squashing predicate is true for any of the instructions ahead of that given instruction. In order to define an ordering among the instructions, we define a measure function `rbi_measure_fn` that associates a measure with every instruction in the reorder buffer such that the tail has measure one and successive instructions have a measure one greater than the previous instruction. So the instructions with lower measures occur earlier in the program order than instructions with higher measures. The function `measure_fn_rbi` returns the reorder buffer index of the instruction with the given measure. To define the abstraction function, we first define a recursive function

Complete_Squash_rest?_till that completes the instructions and computes the disjunction of the squashing predicates from the tail of the reorder buffer till a given unfinished instruction, as shown in [2]. Complete_committed_in_sb_till is a similar recursive function that completes all the committed store instructions in the store buffer. We can then define the abstraction function by first completing all the committed instructions in the store buffer (they are earlier in the program order that any instruction in the reorder buffer) and then completing all the instructions in the reorder buffer. So we define Complete_till and Squash_rest?_till as shown in [2], and then in constructing the abstraction function ABS, we instantiate the Complete_till definition with the measure of the latest instruction in the reorder buffer. The implementation variable rb_count maintains the number of instructions in the reorder buffer, and hence corresponds to the measure of the latest instruction.

### 3.3 Decomposing the Proof

The proof of the commutativity obligation is split into different cases based on the structure of the synchronization function. In this example, the synchronization function returns zero when the processor is restarted or if the squashing predicate evaluates to true for any of the instructions in the reorder buffer (i.e., Squash_rest?_till(q,rb_count(q)) is true) or if no new instruction is issued. Otherwise it returns one, and we consider each of these cases separately. We discuss proving the commutativity obligation for register file rf and program counter pc only. The proofs for the special register file, mode flag and data memory are similar to that rf, though in the case of data memory, one needs to take into account the additional details regarding the committed instructions in the store buffer. The proof for instruction memory is straight-forward as it does not change at all.

We first consider an easy case in the proof of the commutativity obligation (for rf), that is, when the processor is being restarted in the current cycle (restart_proc is true).

- The processor discards all the executing instructions in the reorder buffer, and sets rb_count and lsu_sb_commit_count to zeros. So Complete_till will be vacuous on the implementation side of the commutativity obligation (the side on which I_step(q,s,i) occurs), and the expression on the implementation side simplifies to rf(I_step(q,s,i)).
- Whenever the processor is being restarted, the instruction at the tail of the reorder buffer is causing the rest of the instructions to be squashed, so Squash_rest?_till(q,1) ought to be true. (Recall that the tail of the reorder buffer has measure one.) We prove this, and then from the definition of Complete_Squash_rest?_till in [2], it follows that the kill? argument is true for all the remaining instructions in the reorder buffer, and hence these do not affect rf. Also, the synchronization function returns zero when the processor is being restarted. So the expression on the specification side of the commutativity obligation simplifies to rf(Complete_till(q,1)).

– We show that the `rf(I_step(q,s,i))` and `rf(Complete_till(q,1))` are indeed equal by expanding the definitions occurring in them and simplifying.

Now assume that `restart_proc` is false. We first postulate certain verification conditions, prove them, and then use them in proving the commutativity obligation. Consider an arbitrary instruction `rbi`. Though the processor executes the instructions in an out-of-order manner, it commits the instructions to the observables only in their program order. This suggests that the effect on `rf` of completing all the instructions till `rbi` is the same in the states `q` and `I_step(q,s,i)`. Similarly, the truth value of the disjunction of the squashing predicates till `rbi` is the same in the states `q` and `I_step(q,s,i)`. This verification condition `Complete_Squash_rest?_till_VC` is shown in 3. This is proved by an induction on `rbi_ms`[3] (the measure corresponding to instruction `rbi`).

3

```
% valid_rb_entry? predicate tests whether rbi is within the
% reorder buffer bounds.
Complete_Squash_rest?_till_VC: LEMMA
FORALL(rbi_ms:rbindex): LET rbi = measure_fn_rbi(q,rbi_ms) IN
((valid_rb_entry?(q,rbi) AND NOT restart_proc?(q,s,i)) IMPLIES (
rf(Complete_till(q,rbi_ms)) = rf(Complete_till(I_step(q,s,i),rbi_ms)) AND
Squash_rest?_till(q,rbi_ms) = Squash_rest?_till(I_step(q,s,i),rbi_ms)))
```

As in the earlier proofs based on completion functions approach [HSG99,HSG98], we decompose the proof of `Complete_Squash_rest?_till_VC` into different cases based on how an instruction makes a transition from its present phase to its next phase. Figure 3 shows the phase transitions for an instruction `rbi` in the reorder buffer (when the processor is not restarted) where the predicates labeling the arcs define the conditions under which those transitions take place. The Figure also shows the three transitions for a new instruction entering the processor pipeline. Having identified these predicates, we prove that those transitions indeed take place in the implementation machine. For example, we prove that an instruction `rbi` in phase *dispatched_lsu* (`D_lsu` in the Figure) goes to *executed_lsu* phase in `I_step(q,s,i)` if `Execute_lsu?` predicate is true, otherwise it remains in *dispatched_lsu* phase.

We now return to the proof of `Complete_Squash_rest?_till_VC` and consider the induction argument (i.e., `rbi_ms` is not equal to 1). The proof outline is as follows:

– Expand the `Complete_till` and the `Squash_rest?_till` definitions on both sides of `Complete_Squash_rest?_till_VC` and unroll the recursive definition of `Complete_Squash_rest?_till` once.
– Consider the first conjunct (i.e., one corresponding to `rf`). The `kill?` argument to `Complete_instr` is `Squash_rest?_till(q,rbi_ms-1)` on the left

[3] Since the measure function is dependent on the tail of the reorder buffer, and since the tail can change during an implementation transition, the measure needs to be adjusted on the right hand side of `Complete_Squash_rest?_till_VC` to refer to the same instruction. This is a detail which we ignore in this paper for the ease of explanation, and use just `rbi_ms`.

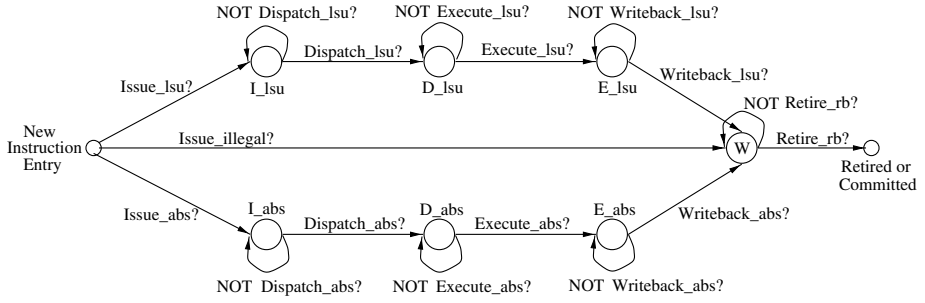


**Fig. 3.** The various phases an instruction can be in and transitions between them (when the processor is not being restarted). Also, the three transitions for an instruction entering the processor are shown.

hand side and `Squash_rest?_till(I_step(q,s,i),rbi_ms-1)` on the right hand side, and these have the same truth value by the induction hypothesis. When it is true, the left hand side reduces to

$$\texttt{rf}(\texttt{Complete\_till}(\texttt{q}, \texttt{rbi\_ms} - 1))$$

and the right hand side to

$$\texttt{rf}(\texttt{Complete\_till}(\texttt{I\_step}(\texttt{q}, \texttt{s}, \texttt{i}), \texttt{rbi\_ms} - 1))$$

which are equal by the induction hypothesis. When it is false, the proof proceeds as in our earlier work [HSG99]. We consider the possible phases `rbi` can be in and whether or not, it makes a transition to its next phase. Assume `rbi` is in *dispatched_abs* phase and the predicate `Execute_abs?` is true. Then, in `I_step(q,s,i)`, `rbi` is in *executed_abs* phase. By the definition of `Complete_instr`, the left hand side of the verification condition simplifies to `rf(Action_dispatched_abs(Complete_till(q,rbi_ms-1),rbi_ms))` and the right hand side reduces to `rf(Action_executed_abs(Complete_till (I_step(q,s,i),rbi_ms-1),rbi_ms))`. The proof now proceeds by expanding these "Action" function definitions, using the necessary invariant properties and simplifying. The induction hypothesis will be used to infer that the register file contents in the two states `Complete_till(q,rbi_ms-1)` and `Complete_till(I_step(q,s,i),rbi_ms-1)` are equal, as those two terms appear when the "Action" definitions are expanded. Overall, the proof decomposes into 14 cases for the seven phases `rbi` can be in.

– Consider the second conjunct of `Complete_Squash_rest?_till_VC`. Using the induction hypothesis, this reduces to showing that the two predicates `Squash_rest?_instr(Complete_till(q,rbi_ms-1),rbi_ms)` and `Squash_rest?_instr(Complete_till(I_step(q,s,i),rbi_ms-1),rbi_ms)` have the same truth value. This proof again proceeds as before by a case analysis on the possible phases `rbi` can be in and whether or not, it makes a transition

to its next phase. The proof again decomposes into 14 cases for the seven phases `rbi` can be in.

For the program counter, however, it is not possible to relate its value in states `q` and `I_step(q,s,i)` by considering the effect of instructions *one at a time in their program order* as was done for `rf`. This is because `I_step` updates `pc` if a new instruction is fetched, either by incrementing it or by updating it according to the speculated branch target address, but this new instruction is the latest one in the program order. However, if the squashing predicate is true for any of the executing instructions in the reorder buffer, then completing that instruction modifies the `pc` with a higher precedence, and the `pc` ought to be modified in the same way in both `q` and `I_step(q,s,i)`. This observation suggests a verification condition on `pc`, shown in 4. This verification condition is again proved by an induction on `rbi_ms`, and its proof is decomposed into 14 cases based on the instruction phase transitions as in the earlier proofs.

```
pc_remains_same_VC: LEMMA                                                    4
FORALL(rbi_ms:rbindex): LET rbi = measure_fn_rbi(q,rbi_ms) IN
 (valid_rb_entry?(q,rbi) AND NOT restart_proc?(q,s,i) AND
        Squash_rest?_till(q,rbi_ms)) IMPLIES
 pc(Complete_till(q,rbi_ms)) = pc(Complete_till(I_step(q,s,i),rbi_ms))
```

Now we come to the proof of the commutativity obligation, where we use the above lemmas after instantiating them with `rb_count`. We consider the different remaining cases in the definition of the synchronization function in order—`Squash_rest?_till(q,rb_count(q))` is true, no new instruction is issued or the three transitions corresponding to a new instruction being issued as shown in Figure 3.

- When `Squash_rest?_till(q,rb_count(q))` is true, the `kill?` argument for the new instruction fetched (if any) will be true in `I_step(q,s,i)` since `Squash_rest?_till` has the same truth value in states `q` and `I_step(q,s,i)`. Hence on the implementation side of the commutativity obligation, there is no new instruction executed. On the specification side, the synchronization function returns zero, so `A_step_new` is vacuous. The proof can then be accomplished using `Complete_Squash_rest?_till_VC` (for the register file) and `pc_remains_same_VC` (for the program counter).
- The proof when no new instruction is issued or when one is issued is similar to the proof in our earlier work [HSG99]. For example, if the issued instruction is in *issued_lsu* phase in `I_step(q,s,i)`, then we have to prove that completing this instruction according to `Action_issued_lsu` has the same effect on the observables as executing a specification machine transition.

**Correctness of the feedback logic:** Whenever there are data dependencies among the executing instructions, the implementation keeps track of them and forwards the results of the execution to all the waiting instructions. The correctness of this feedback logic, both for the register file and the data memory, is expressed in a similar form as in our earlier work [HSG99]. For example, a `load` instruction obtains the value from the store buffer if there is an entry with the

matching address (using associative search), otherwise it reads the value from the data memory. Consider the value obtained when all the instructions ahead of the `load` instruction are completed, and then the data memory is read. This value and the value returned by the feedback logic ought to be equal. The verification condition for the correctness of the feedback logic for data memory is based on the above observation. It will be used in the proof of the commutativity obligation and the proof of this verification condition itself is based on certain invariants.

**Invariants needed:** Many of the invariants needed like the exclusiveness and the exhaustiveness of instruction phases, and the invariants on the feedback logic for the register file and data memory are similar to the ones needed in our earlier work [HSG99]. We describe below one invariant that was not needed in the earlier work.

The LS Unit executes the `load` and `store` instructions in their program order. These instructions are stored in their program order in the reservation stations in the LS Unit and in the store buffer. It was necessary to use these facts during the proof and it was expressed as follows (for the reservation stations in LS Unit): Let `rsi1` and `rsi2` be two instructions in the reservation stations in the LS Unit. `rsi1_ptr` and `rsi2_ptr` point to the reorder buffer entries corresponding to these instructions respectively. Let `lsu_rsi_measure_fn` be a measure function defined on the LS Unit reservation station queue similar to `rbi_measure_fn`. If `rsi1` has a lower/higher measure than `rsi2` according to `lsu_rsi_measure_fn`, then `rsi1_ptr` has a lower/higher measure than `rsi2_ptr` according to a `rbi_measure_fn`.

**PVS proof effort organization:** This exercise was carried out in four phases. In the first phase, we "extrapolated" certain invariants and properties from the earlier work, and this took 27 person hours. In the second phase, we formulated and proved the invariants and certain other properties on the store buffer, and this took 54 person hours. In the third phase, we formulated and proved all the verification conditions about the observables and the commutativity obligation, and this took 131 person hours. In the fourth phase, we proved the necessary invariants about the feedback logic and its correctness, and this took 53 person hours. So the entire proof was accomplished in 265 person hours.

**Related work:** There is one other reported work on formally verifying the correctness of a pipelined processor of comparable complexity. In [SH99], Sawada and Hunt construct an explicit intermediate abstraction in the form of a table called MAETT, express invariant properties on this and prove the final correctness from these invariants. They report taking 15 person months. Also, their approach is applicable to fixed size instantiations of the design only. Various other approaches have been proposed to verify out-of-order execution processors recently [McM98,PA98,JSD98,BBCZ98,CLMK99,BGV99], but none of these

have been so far demonstrated on examples with a similar set of features as we have handled.

## 4 Experimental Results and Concluding Remarks

| Example verified | Effort spent doing the proof |
|---|---|
| `EX1.1` and `EX1.2` | 2 person months |
| `EX3.1` | 13 person days |
| `EX2.1` | 19 person days |
| `EX3.2` | 7 person days |
| `EX2.2` | 34 person days |

**Table 1.** Examples verified and the effort needed.

We have applied our methodology to verify six example processors exhibiting a wide variety of implementation issues, and implemented our methodology in PVS [ORSvH95]. Our results to date are summarized in Table 1. This table summarizes the manual effort spent on each of the examples, listing them in the order we verified them. The first entry includes the time to learn PVS[4]. Each verification effort built on the earlier efforts, and reused some the ideas and the proof machinery.

The processor described in this paper is listed as `EX2.2`. In contrast, `EX1.1` is a five stage pipeline implementing a subset of the DLX architecture, `EX1.2` is a dual-issue version of the same architecture, and `EX2.1` is a processor with a reorder buffer and only arithmetic instructions. We also considered two examples that allowed out-of-order completion of instructions: `EX3.1` allowed certain arithmetic instructions to bypass certain other arithmetic instructions when their destinations were different, and `EX3.2` implemented Tomasulo's algorithm without a reorder buffer and with arithmetic instructions only.

In conclusion, the completion functions approach can be used effectively to verify a wide range of processors against their ISA-level specifications. We have articulated a systematic procedure by which a designer can formulate a very intuitive set of completion functions that help define the abstraction function, and then showed how such a construction of the abstraction function leads to decomposition of the proof of the commutativity obligation. We have also presented how the designer can systematically address details such as exceptions and feedback logic. Design iterations are also greatly facilitated by the completion functions approach due to the incremental nature of the verification, as changes to a pipeline stage do not cause ripple-effects of changes across the whole specification; global re-verification can be avoided because of the layered

[4] By the first author who did all the verification work.

nature of the verification conditions. Our future work will be directed at overcoming the current limitations of the completion functions approach, by seeking ways to automate invariant discovery, especially pertaining to the control logic of processors.

## References

[BBCZ98] Sergey Berezin, Armin Biere, Edmund Clarke, and Yunshan Zu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In Gopalakrishnan and Windley [GW98], pages 369–386.

[BGV99] Randal Bryant, Steven German, and Miroslav Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In Halbwachs and Peled [HP99], pages 470–482.

[CLMK99] Byron Cook, John Launchbury, John Matthews, and Dick Kieburtz. Formal verification of explicitly parallel microprocessors. In Pierre and Kropf [PK99], pages 23–36.

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

[HGS99] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas. A proof of correctness of a processor implementing Tomasulo's algorithm without a reorder buffer. In Pierre and Kropf [PK99], pages 8–22.

[Hos99] Ravi Hosabettu. The Completion Functions Approach homepage, 1999. At address http://www.cs.utah.edu/~hosabett/cfa.html.

[HP99] Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

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

[HSG99] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In Halbwachs and Peled [HP99], pages 47–59.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, BC, Canada, June/July 1998. Springer-Verlag.

[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.

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

[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.

[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.

[PK99] Laurence Pierre and Thomas Kropf, editors. *Correct Hardware Design and Verification Method, CHARME '99*, volume 1703 of *Lecture Notes in Computer Science*, Bad Herrenalb, Germany, September 1999. Springer-Verlag.

[SH99] J. Sawada and W.A. Hunt, Jr. Results of the verification of a comples pipelined machine model. In Pierre and Kropf [PK99], pages 313–316.