A Proof of Correctness of a Processor Implementing Tomasulo's Algorithm Without a Reorder Buffer*

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. 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. However, its applicability depends on the fact that the implementation "commits" the unfinished instructions in the pipeline in program order. In this paper, we extend the completion functions approach when this is not true and demonstrate it on an implementation of Tomasulo's algorithm without a reorder buffer. The approach leads to an elegant decomposition of the proof of the correctness criterion, does not involve the construction of an explicit intermediate abstraction, makes heavy use of an automatic case-analysis strategy based on decision procedures and rewriting, and addresses both safety and liveness issues.

1 Introduction

For formal verification to be successful in practice, not only is it important to raise the level of automation 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 in verification 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

* The first and second authors were supported in part by NSF through Grant no. CCR-9800928. The third author was supported in part by NASA contract NAS1-20334 and ARPA contract NASA-NAG-2-891 (ARPA Order A721).

checking provides a way of decomposing the overall proof and reducing the effort for an individual model checker run, a practical methodology for decision 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[1] 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, the desired effect on the observables of completing that instruction, assuming those ahead of it in the program order are completed. 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 (such as the lemma on the correctness of the feedback logic) 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.
- The verification avoids the construction of an explicit intermediate abstraction as well as the large amount of manual effort required to construct it.

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. The proof decomposition that this method achieves and the verification conditions generated in the DLX example is illustrated in Figure 1.

Later, we extended the methodology to verify a truly out-of-order execution processor with a reorder buffer [HSG99]. We observed that regardless of how many instructions are pending in the reorder buffer, the instructions can only be in one of a few (small finite number) distinct states and exploited this fact to provide a single compact parameterized completion function applicable to all the pending instructions in the reorder buffer. The proof was decomposed on the basis of how an instruction makes a transition from its present state to the next state.

However, the applicability of the completion functions approach depends on the fact that the implementation "commits" the unfinished instructions in the

[1] Our correctness criteria is based on using an abstraction function, as most others.

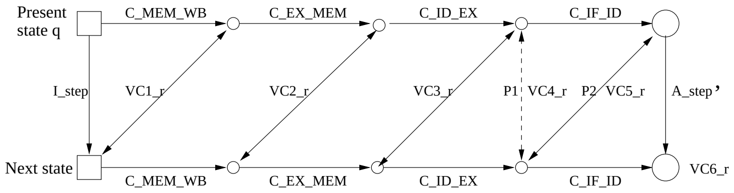

Fig. 1. The proof decomposition in the DLX example using the completion functions approach (C_... are the completion functions for the various unfinished instructions).

pipeline in program order. The abstraction function is defined by composing the completion functions of the unfinished instructions in the program order too. Because of this, it is possible to relate the effect of completing instructions one at a time in the present and the next states and incrementally build the proof of the commutative diagram (See Figure 1). Also, one can provide for every unfinished instruction, an "abstract" state where the instructions ahead of it are completed. This fact is useful in expressing the correctness of the feedback logic. If instructions were to commit out-of-order, it is not possible to use these ideas.

A processor implementing Tomasulo's algorithm without a reorder buffer executes instructions in the data-flow order, possibly committing them to the register file in an out-of-order manner. Hence, the basic premise of the completion functions approach—that instructions commit in the program order—is not true in this case. The implementation maintains the identity of the latest instruction writing a particular register. Those instructions issued earlier and not the latest ones to write their respective destinations, on completing their execution, only forward the results to other waiting instructions but do not update the register file. Observe that it is difficult to support branches or exceptions in such an implementation. (In an implementation supporting branches or exceptions, the latest instruction writing a register can not be easily determined.)

In this paper, we extend the completion functions approach to be applicable in such a scenario. Instead of defining the completion function to directly update the observables, we define it to return the value an instruction computes in the various states. The completion function for a given instruction recursively completes the instructions it is dependent on to obtain its source values. The abstraction function is defined to assign to a register the value computed by the latest instruction writing that register. We show that this modified approach leads to a decomposition of the overall proof of correctness, and we make heavy use of an automatic case-analysis strategy in discharging the different obligations in the decomposition. The proof does not involve the construction of an explicit intermediate abstraction. Finally, we address the proof of liveness properties too.

The rest of the paper is organized as follows: In Section 2, we describe our processor model. Section 3 describes our correctness criteria. This is followed by the proof of correctness in Section 4. We compare our work with others in Section 5 and finally provide the conclusions.

2 Processor Model

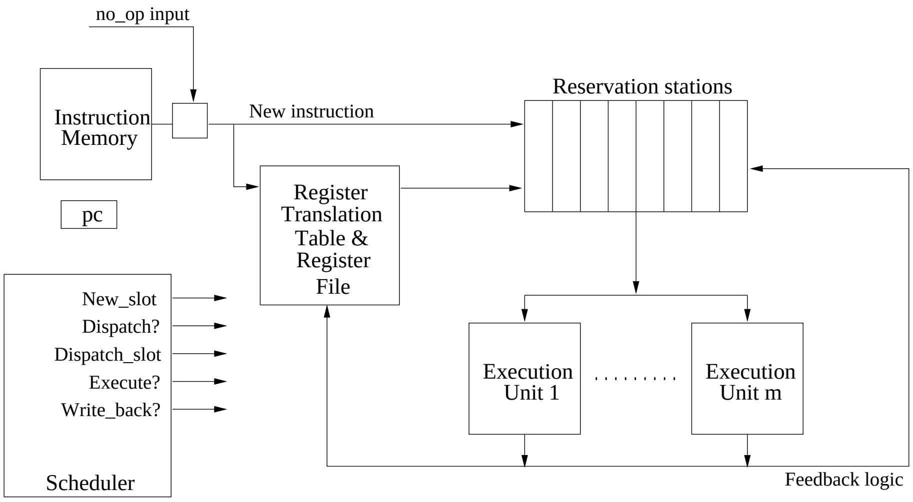

Fig. 2. The block diagram model of our implementation

Figure 2 shows the model of an out-of-order execution processor implementing Tomasulo's algorithm without a reorder buffer used in this paper. The model has `z` reservation stations 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 register translation table (RTT) maintains the identity of the latest pending instruction writing a particular register (the identity is a "tag"—in this case, the reservation station index). A scheduler controls the movement of the instructions through the execution pipeline (such as being dispatched, executed etc) and its behavior is modeled in the form of axioms (instead of a concrete implementation). 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 a free reservation station for it (`New_slot`). No instruction is issued if all the reservation stations are occupied or if `no_op` is asserted. The RTT entry corresponding to 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 tags of the instructions providing the source operands is maintained in the reservation station allocated to the instruction. An issued instruction monitors the execution units to see if they produce the values it is waiting for, by comparing the tags it is waiting on with the tags of the instructions producing the result. An instruction can be *dispatched* when its source operands are ready and the corresponding execution unit is free. `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 reservation station index from where to dispatch. Dispatched instructions get *executed* after a non-deterministic amount of time as determined by the scheduler output `Execute?`. At a time determined by the `Write_back?` output of the scheduler, an execution unit writes back its result which will be forwarded to other waiting instructions. A register updates its value with this result only if its RTT entry matches the tag of the instruction producing the result and then clears its RTT entry. Finally, when an instruction is written back, its reservation station is freed.

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 3(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 (due to Hoare [Hoa72]) shown in Figure 3(b) is proved on a single implementation machine transition (and a certain other condition discussed in the next paragraph holds).

The criterion in Figure 3(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* obliga-

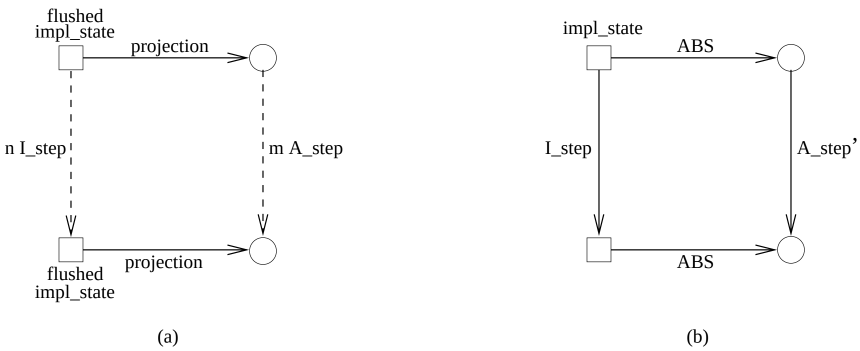

Fig. 3. Pipelined microprocessor correctness criteria

tion). 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 3(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 3(b) and prove that it is closed under `I_step`.

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 `next(q,s,i)` the "next state" after an implementation transition. "Primed" variables will be used to refer to the value of a given variable in the next state. Also, we identify an instruction in the processor by its reservation station index (i.e., instruction `rsi` means instruction at reservation station index `rsi`). When the instruction in question is clear from the context (say `rsi`), we use just `rs_op` to refer to its opcode instead of `rs_op(q)(rsi)`. (`rs_op`′ will refer to `rs_op(next(q,s,i))(rsi)`). The 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 three following possible states inside the processor—issued, dispatched or executed. (Once written back, it is no longer present in the processor). We formulate predicates describing an instruction in each of these states and specify the value an instruction computes in each of these states. The definition of the completion function is shown in $\boxed{1}$.

```
% state_I : implementation state type; rsindex : reservation station   1
% index type; value : type of the data computed by an instruction.
Complete_instr(q:state_I,rsi:rsindex): RECURSIVE value =
   IF executed_pred(q,rsi) THEN Value_executed(q,rsi)
   ELSIF dispatched_pred(q,rsi) THEN Value_dispatched(q,rsi)
   ELSIF issued_pred(q,rsi) THEN
   % Value_issued(q,rsi) expanded to highlight the recursive call.
   % alu is an uninterpreted function. ``rs_op'' is the opcode.
      alu(rs_op(q)(rsi),
          IF rs_src_ptr1(q)(rsi) = 0 THEN
             rs_src_value1(q)(rsi)
          ELSE Complete_instr(q,rs_src_ptr1(q)(rsi)) ENDIF,
          ``Second operand -- similar definition'')
   ELSE default_value ENDIF
   MEASURE rs_instr_num(q)(rsi)
```

In this implementation, when an instruction is in the executed state, the result value is available in `eu_result` field of the execution unit, so `Value_executed` returns this value. We specify `Value_dispatched` along the same lines. When an instruction is in the issued state, it may be waiting for its source operands to get ready. In determining the value computed by such an instruction, we need the source operands which we specify as follows: When `rs_src_ptr1` is zero, the first source operand is ready and its value is available in `rs_src_value1`, otherwise its value is obtained by completing the instruction it is waiting on (`rs_src_ptr1` points to that instruction). Similarly the second source operand is specified.

To specify the completion function, we added three auxiliary variables. The first one maintains the index of the execution unit an instruction is dispatched to. Since the completion function definition is recursive, one needs to provide a measure function to show that the function is well-defined ; the other two auxiliary variables are for this purpose. We should prove that instructions producing the source values for a given instruction `rsi` have a lower measure than `rsi`. So we assign a number `rs_instr_num` to every instruction that records the order in which it is issued and this is used as the measure function. (The counter that is used in assigning this number is the third auxiliary variable).

4.2 Constructing the abstraction function

The register translation table maintains the identity of the latest pending instruction writing a particular register. The abstraction function is constructed by updating every register with the value obtained by completing the appropriate pending instruction, as shown in $\boxed{2}$. The synchronization function returns zero if `no_op` input is asserted or if there is no free reservation station to issue an instruction, otherwise returns one.

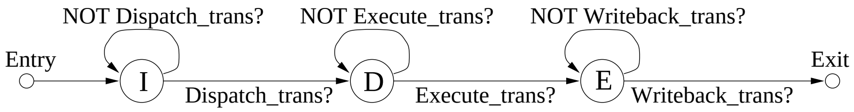

Fig. 4. The various states an instruction can be in and transitions between them, I: issued, D: dispatched, E: executed

```
% If the ``rtt'' field for a given register is zero, then it is    2
% not updated, otherwise complete the instruction pointed to by
% ``rtt'' and update the register with that value.
Complete_all(q:state_I): state_I =
   q WITH [ (rf) := LAMBDA(r:reg):
                      IF rtt(q)(r) = 0 THEN rf(q)(r)
                      ELSE Complete_instr(q,rtt(q)(r)) ENDIF ]

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

4.3 Proof decomposition

We first prove a lemma that characterizes the value an instruction computes and then use it in the proof of the commutative diagram. Consider an arbitrary instruction `rsi`. We claim that the value an instruction computes (as given by `Complete_instr`) is the same whether in state `q` or in state `next(q,s,i)`, as long as the instruction is valid in these states. (Intuitively, an instruction is valid as long as it has not computed and written back its result.) This is shown as lemma `same_result` in $\boxed{3}$. We prove this by induction on `rsi` (induction with a measure function as explained later).

```
% rs_valid means the instruction is valid.                          3
same_result: LEMMA
    FORALL(rsi:rsindex):
    (rs_valid(q)(rsi) AND rs_valid(next(q,s,i))(rsi))
         IMPLIES
    Complete_instr(q,rsi) = Complete_instr(next(q,s,i),rsi)
```

We generate the different cases of the induction argument (as will be detailed shortly) based on how an instruction makes a transition from its present state to its next state. This is shown in Figure 4 where we have identified the conditions under which an instruction changes its state. For example, we identify the predicate `Dispatch_trans?(q,s,i,rsi)` which takes the instruction `rsi` 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 `rsi`. Similarly other "trans" predicates are defined.

Having defined these predicates, we prove that they indeed cause instructions to take the transitions shown. Consider a valid instruction `rsi` in the issued state i.e., `issued_pred(q,rsi)` holds. We prove that if `Dispatch_trans?(q,s,i,rsi)` is true, then after an implementation transition, `rsi` will be in dispatched state (i.e., `dispatched_pred(next(q,s,i),rsi)` is true) and remains valid. (This is shown as a lemma in $\boxed{4}$.) Otherwise (if `Dispatch_trans?(q,s,i,rsi)` is false), we prove that `rsi` remains in the issued state in `next(q,s,i)` and remains valid. There are three other similar lemmas for the other transitions. The sixth lemma is for the case when an instruction `rsi` in the executed state is written back. It states that `rsi` is no longer valid in `next(q,s,i)`.

```
issued_to_dispatched: LEMMA                                          4
    FORALL(rsi:rsindex):
    (rs_valid(q)(rsi) AND issued_pred(q,rsi) AND
                          Dispatch_trans?(q,s,i,rsi))
       IMPLIES
    (dispatched_pred(next(q,s,i),rsi) AND rs_valid(next(q,s,i),rsi))
```

Now we come back to the details of the `same_result` lemma. In proving this lemma for an instruction `rsi`, one needs to assume that the lemma holds for the two instructions producing the source values for `rsi` (Details will be presented later). So we do an induction on `rsi` with `rs_instr_num` as the measure function. As explained earlier in Section 4.1, instructions producing the source values (`rs_src_ptr1` and `rs_src_ptr2` when non-zero) have a lower measure than `rsi`. The induction argument is based on a case analysis on the possible state `rsi` is in, and whether or not it makes a transition to its next state. Assume the instruction `rsi` is in issued state. We prove the induction claim in the two cases—`Dispatch_trans?(q,s,i,rsi)` is true or false—separately. (The proof obligation for the first case is shown in $\boxed{5}$.) We have similar proof obligations for `rsi` being in other states. In all, the proof decomposes into six proof obligations.

```
% One of the six cases in the induction argument.                    5
issued_to_dispatched_induction: LEMMA
    FORALL(rsi:rsindex):
    (rs_valid(q)(rsi) AND issued_pred(q,rsi) AND
    Dispatch_trans?(q,s,i,rsi) AND Induction_hypothesis(q,s,i,rsi))
         IMPLIES
    Complete_instr(q,rsi) = Complete_instr(next(q,s,i),rsi)
```

We sketch the proof of `issued_to_dispatched_induction` lemma. We refer to the goal that we are proving—`Complete_instr( ... ) = Complete_instr( ... )`—as the consequent. We expand the definition of the completion function corresponding to `rsi` on both sides of the consequent. In `q`, `rsi` is in the issued state and in `next(q,s,i)`, it is the dispatched state—this follows from the `issued_to_dispatched` lemma. After some rewriting and simplifications in PVS, the left hand side of the consequent simplifies to `Value_issued(q,rsi)` and the right hand side simplifies to `Value_dispatched(next(q,s,i),rsi)`. (The proofs of all the obligations are similar till this point. After this point, it depends on the particular obligation being proved since different invariants are needed for the different obligations.) Proof now proceeds by expanding the definitions of `Value_issued` and `Value_dispatched`, using the necessary invariants and simplifying. We use the PVS strategy `apply (then* (repeat (lift-if))` (bddsimp) (ground) (assert)) to do the simplifications by automatic case-analysis (many times, simply assert will do).

We illustrate the proof of another lemma issued_remains_induction (shown in $\boxed{6}$) in greater detail pointing out how the feedback logic gets verified. As above, the proof obligation reduces to showing that Value_issued(q,rsi) and Value_issued(next(q,s,i),rsi) are the same. (The definition of Value_issued is shown in $\boxed{1}$.) This can be easily proved once we show that the source values of rsi as defined by op_val1 (and a similar op_val2) remain same, whether in q or in next(q,s,i). Proving this lemma op_val1_same (and a similar op_val2_same) establishes the correctness of the feedback logic.

```
% Value of the first operand.                                        6
op_val1(q:state_I,rsi:rsindex): value =
    IF rs_src_ptr1(q)(rsi) = 0 THEN rs_src_value1(q)(rsi)
    ELSE Complete_instr(q,rs_src_ptr1(q)(rsi)) ENDIF

op_val1_same: LEMMA
    FORALL(rsi:rsindex):
    (rs_valid(q)(rsi) AND issued_pred(q,rsi) AND
    NOT Dispatch_trans?(q,s,i,rsi) AND Induction_hypothesis(q,s,i,rsi))
           IMPLIES
    op_val1(q,rsi) = op_val1(next(q,s,i),rsi)

issued_remains_induction: LEMMA
    FORALL(rsi:rsindex):
    (rs_valid(q)(rsi) AND issued_pred(q,rsi) AND
    NOT Dispatch_trans?(q,s,i,rsi) AND Induction_hypothesis(q,s,i,rsi))
           IMPLIES
    Complete_instr(q,rsi) = Complete_instr(next(q,s,i),rsi)
```

In proving op_val1_same lemma, there are three cases. Consider the case when rs_src_ptr1 is zero. We then show that rs_src_ptr1′ is zero and rs_src_value1 is the same as rs_src_value1′. Consider the case when rs_src_ptr1 is non-zero. rs_src_ptr1′ may or may not be zero. If rs_src_ptr1′ is zero, then it implies that in the current cycle, the instruction pointed to by rs_src_ptr1 completes its execution and forwards its result to rsi. So it is easy to prove rs_src_value1′ (the value actually written back in the implementation) is the same as the expected value Complete_instr(q,rs_src_ptr1(q)(rsi)). If rs_src_ptr1′ is non-zero, then one can conclude from the induction hypothesis that rs_src_ptr1 computes the same value in q and in next(q,s,i).

Proving the commutative diagram Consider the case when no new instruction is issued in the current cycle, that is, the synchronization function returns zero. The commutative diagram obligation in this case is shown in $\boxed{7}$.

```
% sch_rs_slot (i.e., scheduler output New_slot) is valid means no      7
% free reservation stations.
commutes_no_issue: LEMMA
    (no_op?(i) OR rs_valid(q)(sch_rs_slot(s)))
        IMPLIES
    rf(ABS(q)) = rf(ABS(next(q,s,i)))
```

We expand the definition of ABS (shown in $\boxed{2}$) and consider a particular register r. This again leads to three cases as in the correctness of op_val1_same. Consider the case when rtt (i.e., rtt(q)(r)) is zero. We then show that rtt′ is zero too and the values of register r match in q and next(q,s,i). Consider the case when rtt is non-zero. rtt′ may or may not be zero. If rtt′ is zero, then it implies that in the current cycle, the instruction pointed to by rtt completes its execution and writes its result to r. It is easy to show that this value written into r is the same as the expected value Complete_instr(q,rtt(q)(r)). If rtt′ is non-zero, then we use same_result lemma to conclude that the same value is written into r in q and next(q,s,i).

The case when a new instruction is issued is similar to the above except when r is the destination register of the instruction being issued. We show that in state next(q,s,i), the new instruction is in issued state, its operands as given by op_val1 and op_val2 equal the ones given by the specification machine and the value written into r by the implementation machine equals the value given by specification machine.

The program counter pc is incremented whenever an instruction is fetched. This is the only way pc is modified. So proving the commutative diagram for pc is simple. The commutative diagram proof for the instruction memory is trivial since it is not modified at all.

The invariants needed We describe in this section *all* the seven invariants needed by our proof. We do not have a uniform strategy for proving all these invariants but we use the automatic case-analysis strategy shown earlier to do the simplifications during the proofs.

- Two of invariants are related to rs_instr_num and instr_counter, the auxiliary variables introduced for defining a measure for every instruction. The first invariant states that the measure of any instruction (rs_instr_num) is less than the running counter (instr_counter). The second one states that for any instruction, if the source operands are not ready, then the measure of the instructions producing the source values is less than the measure of the instruction. The need for these was realized when we decided to introduce the two auxiliary variables mentioned above.
- Two other invariants are related to rs_exec_ptr, the auxiliary variable that maintains the execution unit index an instruction is dispatched to. The first invariant states that, if rs_exec_ptr is non-zero, then that execution unit is busy and its tag (which records the instruction executing in the unit) points to the instruction itself. The second invariant states that, whenever an execution unit is busy, the instruction pointed to by its tag is valid and that

instruction's rs_exec_ptr points to the execution unit itself. These invariants are very similar to ones we needed in an earlier verification effort [HSG99].

- Two other invariants characterize when an instruction is valid. The first one states that for any register, the instruction pointed to by rtt is valid. The second one states that for any given instruction, the instructions pointed to by rs_src_ptr1 and rs_src_ptr2 are valid. The final invariant we needed was that rs_exec_ptr for any instruction is non-zero if and only if rs_disp? (a boolean variable that says whether or not an instruction is dispatched) is true. The need for these three invariants was realized during the proofs of other lemmas/invariants.

PVS proof timings: The proofs of all the lemmas and the invariants discussed so far takes about 500 seconds on a 167 MHz Ultra Sparc machine. [2]

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. In fact, most of them are related to the "instruction state" transitions shown in Figure 4 and the additional obligations needed (not proved earlier) takes only about 15 seconds on a 167 MHz Ultra Sparc machine. We now provide a sketch of the high level argument which is being formalized in PVS.
Proof sketch: The processor is flushed if for all registers r, rtt(q)(r) = 0.

- First, we show that "any valid instruction in the dispatched state eventually goes to the executed state and be valid" and "any valid instruction in the executed state eventually gets written back and its reservation station will be freed". Consider a valid instruction rsi in the dispatched state. If in state q, Execute_trans?(q,s,i,rsi) is true, then rsi goes to the executed state in next(q,s,i) and remains valid (refer to Figure 4). Otherwise it continues to be in the dispatched state and remains valid. We observe that when rsi 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 rsi 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,rsi) will be true) and in next(q,s,i), the instruction will be in the executed state and be valid. By a similar argument, it eventually gets written back and the reservation station gets freed.

[2] The manual effort involved in doing the proofs was one person week. The authors had verified a processor with a reorder buffer earlier [HSG99] and most of the ideas/proofs carried over to this example.

- Second, we show that "every busy execution unit eventually becomes free and stays free until an instruction is dispatched on it". This follows from the observation that whenever an execution unit is busy, the instruction occupying it is in the dispatched/executed state and that such an instruction eventually gets written back (first observation above).
- Third, we show that "a valid instruction in the issued state will eventually go to the dispatched state and be valid". Here, the proof is by induction (with `rs_instr_num` as the measure) since an arbitrary instruction `rsi` could be waiting for two previously issued instructions to produce its source values. Consider a valid instruction `rsi` in the issued state. If the source operands of `rsi` are ready, then we observe that the scheduler inputs that determine dispatching remain asserted as long as `rsi` is not dispatched. Busy execution units eventually get free and remain free until an instruction is dispatched on it (second observation above). So by a fairness assumption on the scheduler, `rsi` eventually gets dispatched. If a source operand is not ready, then the instruction producing it has a lower measure. By the induction hypothesis, it eventually goes to the dispatched state, eventually gets written back (first observation) forwarding the result to `rsi`. By a similar argument as above, `rsi` eventually gets dispatched.
- Finally, we show that "the processor eventually gets flushed". We observe that every valid instruction in the processor eventually gets written back freeing its reservation stations (third and first observations). Since no new instructions are being inserted, free reservation stations remain free. Whenever `rtt(q)(r)` is non-zero, it points to an occupied reservation station. Since, eventually all reservation stations get free, all `rtt` entries become zero and the processor is flushed.

5 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. In particular, prior to our work, one theorem prover based and three model checking based verifications of a similar example—processor implementing Tomasulo's algorithm without a reorder buffer—have been carried out.

The theorem prover based verification reported in [AP98] is based on *refinement* and the use of "predicted value". They introduce this "predicted value" as an auxiliary variable to help in comparing the implementation against its specification without constructing an intermediate abstraction. However there is no systematic way to generate the invariants and the obligations needed in their approach. And they do not address liveness issues needed to complete the proof.

A model checking based verification of Tomasulo's algorithm is carried out in [McM98]. He uses compositional model checking and aggressive symmetry reductions to manually decompose the proof 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. However the proof is dependent on the configuration of the processor (number of reservation stations etc) and also on the actual arithmetic operators.

Another verification of Tomasulo's algorithm is reported in [BBCZ98] where they combine symbolic model checking with uninterpreted functions. They introduce a data structure called reference file for representing the contents of the register file. While they abstract away from the data path, the verification is for a fixed configuration of the processor and they is no decomposition of the proof.

Yet another verification based on assume-guarantee reasoning and refinement checking is presented in [HQR98]. The proof is decomposed by providing the definitions of suitable "abstract" modules and "witness" modules. However the proof can be carried out for a fixed small configuration of the processor only.

Finally, verification of a processor model implementing Tomasulo's algorithm with a reorder buffer, exceptions and speculative execution is carried out in [SH98]. 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.

6 Conclusion

We have showed in this paper how to extend the completion functions approach to be applicable in a scenario where the instructions are committed out-of-order and illustrated it on a processor implementation of Tomasulo's algorithm without a reorder buffer. Our approach lead to an elegant decomposition of the proof based on the "instruction state" transitions and did not involve the construction of an intermediate abstraction. The proofs made heavy use of an automatic case-analysis strategy and addressed both safety and liveness issues.

We are currently developing a PVS theory of the "eventually" temporal operator to mechanize the liveness proofs presented here. We are also working on extending the completion functions approach further to verify a detailed out-of-order execution processor (with a reorder buffer) involving branches, exceptions and speculative execution.

References

[AP98] T. Arons and A. Pnueli. Verifying Tomasulo's algorithm by refinement. Technical report, Weizmann Institute, 1998.

[BBCZ98] Sergey Berezin, Armin Biere, Edmund Clarke, and Yunshan Zu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design, FMCAD '98*, volume 1522 of *Lecture Notes in Computer Science*, pages 369–386, Palo Alto, CA, USA, November 1998. Springer-Verlag.

[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 *Lecture Notes in Computer Science*, pages 187–201, Palo Alto, CA, November 1996. Springer-Verlag.

[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 *Lecture Notes in Computer Science*, pages 203–222, Bad Herrenalb, Germany, September 1994. Springer-Verlag.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representations. In *Acta Informatica*, volume 1, pages 271–281, 1972.

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

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

[HQR98] Thomas Henzinger, Shaz Qadeer, and Sriram Rajamani. You assume, we guarantee: Methodology and case studies. In Hu and Vardi [HV98], pages 440–451.

[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. 1999. Accepted for publication in the Conference on Computer Aided Verification, Trento, Italy.

[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.

[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.

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