Decomposing the Proof of Correctness of Pipelined Microprocessors*

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. We present a systematic approach to decompose and incrementally build 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 addition to avoiding term-size and case explosion problem that limits the pure *flushing* approach, our method helps localize errors, and also handles stages with iterative loops. The technique is illustrated on a pipelined and a superscalar pipelined implementations of a subset of the DLX architecture. It has also been applied to a processor with out-of-order execution.

1 Introduction

Many modern microprocessors employ radical optimizations such as superscalar pipelining, speculative execution and out-of-order execution to enhance their throughput. These optimizations make microprocessor verification difficult in practice. Most approaches to mechanical verification of pipelined processors rely on the following key techniques: First, given a pipelined implementation and a simpler ISA-level specification, they require a suitable abstraction mapping from an implementation state to a specification state and define the correspondence between the two machines using a commutative diagram. Second, they use symbolic simulation to derive logical expressions corresponding to the two paths in the commutative diagram which will then be tested for equivalence. An automatic way to perform this equivalence testing is to use ground decision procedures for equality with uninterpreted functions such as the ones in PVS. This strategy has been used to verify several processors in PVS [5, 4, 15]. Some of the approaches to pipelined processor verification rely on the user providing the definition for the abstraction function. Burch and Dill in [3] observed that the

* This work was done in part when Ravi Hosabettu was visiting SRI International in summer 1997. The work done by the authors at University of Utah was supported in part by DARPA under Contract #DABT6396C0094 (Utah Verifier) and NSF MIP MIP–9321836. The work done by the authors at SRI International was supported in part under NASA Contract NAS1–20334 and ARPA Contract A721/NAG 2–891.

effect of flushing the pipeline, for example by pumping a sequence of NOPs, can be used to automatically compute a suitable abstraction function. Burch and Dill used this *flushing approach* along with a validity checker [9, 1] to automate effectively the verification of pipelined implementations of several processors.

The pure flushing approach has the drawback of making the size of the abstraction function generated and the number of examined cases impractically large for deep and complex superscalar pipelines. To verify a superscalar example using the flushing approach, Burch [2] decomposed the verification problem into three subproblems and suggested a technique which required the user to add some extra control inputs to the implementation and set them appropriately to construct the abstraction function. He also had to fine-tune the validity checker used in the experiment requiring the user to help it with many manually derived case splits. It is unclear how the decomposition of the proof and the abstraction function used in [2] can be reused for verifying other superscalar examples. Another drawback of the pure flushing approach is that it is hard to use for pipelines with indeterminate latency, which can arise if the control involves data-dependent loops or if some part of the processor, such as memory-cache interface, is abstracted away for managing the complexity of the system.

In this paper, we propose a systematic methodology to modularize as well as decompose the proof of correctness of microprocessors with complex pipeline architectures. Called the *completion functions* method, our approach relies on the user expressing the abstraction function in terms of a set of completion functions, one per unfinished instruction. Each completion function specifies the *desired effect* (on the observables) of completing the instruction. Notice that one is not obligated to state *how* such completion would actually be attained, which, indeed, can be very complex, involving details such as squashing, pipeline stalls, and even data dependent iterative loops. Moreover, we strongly believe that a typical designer would have a very clear understanding of the completion functions, and would not find the task of describing them and constructing the abstraction function onerous. In addition to actually gaining from designers' insights, verification based on the completion functions method has a number of other advantages. It results in a natural decomposition of proofs. Proofs build up in a layered manner where the designer actually debugs the last pipeline stage first through a verification condition, and then uses this verification condition as a rewrite rule in debugging the penultimate stage, and so on. Because of this layering, the proof strategy employed is fairly simple and almost generic in practice. Debugging is far more effective than in other methods because errors can be localized to a stage, instead of having to wade through monolithic proofs.

1.1 Related Work

Levitt and Olukotun [10] use an "unpipelining" technique for merging successive pipeline stages through a series of behavior preserving transformations. While unpipelining also results in a decomposition of the proofs, their transformation is performed on the implementation whereas completion functions are defined based on the specification. Their method has the disadvantage that the implementation is verified against itself in the initial steps and that their transformations can get complex for superscalar processors and processors with out-of-order execution. Cyrluk's technique in [6], which has also been applied to a superscalar processor, tackles the term-size and case explosion problem by lazily "inverting the abstraction mapping" to replace big implementation terms with smaller specification terms and using the conditions in the specification terms to guide the proof. Our method contains the complexity of the proof by decomposing the proof of the commutative diagram one stage at a time in a fashion that is closer to the user's intuition about the design. Park and Dill have used aggregation functions [12], which are conceptually similar to completion functions, for distributed cache coherence protocol verification. In [13], Sawada and Hunt used an incremental verification technique to verify a processor with out-of-order execution which we have reverified using our approach. We describe the differences in section 4.5.

2 Correctness Criteria for Processor Verification

The completion functions approach aims to realize the correctness criterion (used in [13, 3]) expressed in Figure 1(a), in a manner that proofs based on it are modular and layered as pointed out earlier.

Figure 1(a) requires that every sequence of `n` implementation transitions which start and end with *flushed* states (i.e., no partially executed instructions) corresponds to a sequence of `m` instructions (i.e., transitions) executed by the specification machine. `I_step` is the implementation transition function and `A_step` is the specification transition function. The `projection` extracts only those implementation state components visible to the specification (i.e. the observables). This criterion is preferred over others that have been used to verify pipelined processors because it corresponds to the intuition that a real pipelined microprocessor starting at a flushed state, running some program and terminating in a flushed state is emulated by a specification machine whose starting and terminating states are in direct correspondence through projection. This criterion can be proved by induction on `n` once the *commutative diagram* condition shown in Figure 1(b) has been proved on a single implementation machine transition. This inductive proof can be constructed once, as we have demonstrated in [8], for arbitrary machines that satisfy the conditions described in the next paragraph. In the rest of the paper, we concentrate on verifying the commutative diagram condition.

Intuitively, Figure 1(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. `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

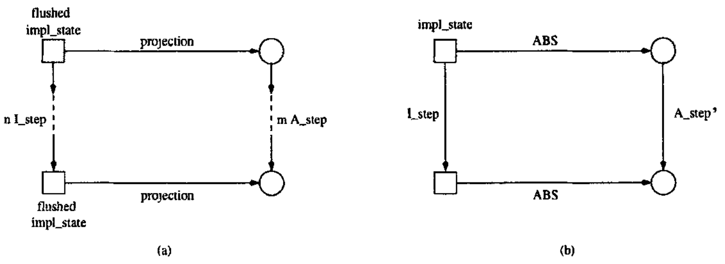

Fig. 1. Pipelined microprocessor correctness criteria

transition from an arbitrary state might correspond to executing in the specification machine zero instruction (*e.g.*, if the implementation machine stalls due to pipeline interlocks) or more than one instruction (*e.g.*, if the implementation machine has multiple pipelines). 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. 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 1(a) is not vacuous. In addition, the user may need to discover *invariants* to restrict the set of `impl_state` considered in the proof of Figure 1(b) and prove that it is closed under `I_step`.

3 The Completion Functions Approach

One way of defining `ABS` is to use a part of the implementation definition, modified, if necessary, to construct an explicit *flush* operation [3, 2] to flush the pipeline. The completion functions approach is based on using an abstraction function that is behaviorally equivalent to flushing but is *not* derived operationally via flushing in our basic approach[1]. Rather, we construct the abstraction function as a composition (followed by a projection) of a sequence of *completion functions* that map an implementation state to an implementation state. Each completion function specifies the *desired effect* on the observables of completing a particular unfinished instruction in the machine leaving all non-observable state components unchanged. The order in which these functions are composed is determined by the program order of the unfinished instructions. The conditions under which each function is composed with the rest, if any, is determined by whether the unfinished instructions ahead of it could disrupt the flow of instructions for example, by being a taken branch or by raising an exception. Observe that one is not required to state how these conditions are actually realised in the implementation. Any mistakes, either in specifying the completion functions or

[1] Later we discuss a hybrid scheme extension that uses operational flushing.

in constructing the abstraction function, might lead to a false negative verification result, but never a false positive.

Consider a very simple four-stage pipeline with one observable state component `regfile` which is shown in Figure 2. The instructions flow down the pipeline with every cycle in order with no stalls, hazards etc. updating the `regfile` in the last stage. (This is unrealistically simple, but we explain how to handle these artifacts in subsequent sections.) The pipeline can contain three unfinished instructions at any time, which are held in the three sets of pipeline registers labeled IF/ID, ID/EX, and EX/WB. The completion function corresponding to an unfinished instruction held in a set of pipeline registers (such as ID/EX) defines how the information stored in those registers are combined to complete that instruction. In our example, the completion functions are `C_EX_WB`, `C_ID_EX` and `C_IF_ID`, respectively. Now the abstraction function, whose effect should be to flush the pipeline, can be expressed as a composition of these completion functions as follows (we omit `projection` here as `regfile` is the only observable state component):

```
ABS(impl_state) = C_IF_ID(C_ID_EX(C_EX_WB(impl_state)))
```

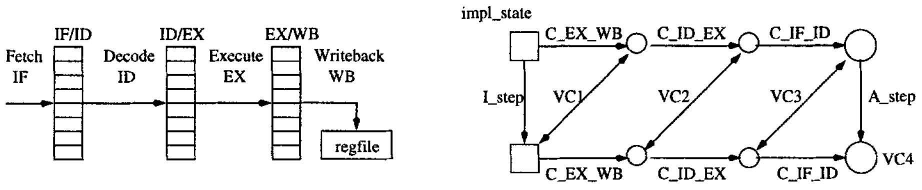

Fig. 2. A simple four stage pipeline and decomposition of the proof under completion functions

This definition of the abstraction function leads to a decomposition of the proof of the commutative diagram for `regfile` as shown in Figure 2, generating the following series of verification conditions, the last one of which corresponds to the complete commutative diagram.

```
VC1: regfile(I_step(impl_state)) = regfile(C_EX_WB(impl_state))
VC2: regfile(C_EX_WB(I_step(impl_state))) =
              regfile(C_ID_EX(C_EX_WB(impl_state)))
VC3: regfile(C_ID_EX(C_EX_WB(I_step(impl_state)))) =
              regfile(C_IF_ID(C_ID_EX(C_EX_WB(impl_state))))
VC4: regfile(C_IF_ID(C_ID_EX(C_EX_WB(I_step(impl_state))))) =
              regfile(A_step(C_IF_ID(C_ID_EX(C_EX_WB(impl_state)))))
```

`I_step` executes *partially* the instructions already in the pipeline as well as a newly fetched instruction. Given this, VC1 expresses the following fact: since `regfile` is updated in the last stage, we would expect that after `I_step` is executed, the contents of `regfile` would be the same as after completing the instruction in the EX/WB registers.

Now consider the instruction in ID/EX. I_step executes it partially as per the logic in stage EX, and then moves the result to the EX/WB registers. `C_EX_WB` can now be used to complete this instruction. This must result in the same contents of `regfile` as completing the instructions held in sets EX/WB and ID/EX of pipeline registers *in that order*. This is captured by VC2. VC3 and VC4 are constructed similarly. Note that our ultimate goal is to prove only VC4, with the proofs of VC1 through VC3 acting as "helpers". Each verification condition in the above series can be proved using a *standard strategy* that involves expanding the outermost function on the both sides of the equation and using the previously proved verification conditions (if any) as rewrite rules to simplify the expressions, followed by automatic case analysis of the boolean terms appearing in the conditional structure of the simplified expressions. Since we expand only the topmost functions on both sides, and because we use the previously proved verification conditions, the sizes of the expressions produced during the proof and the required case analysis are kept in check.

The completion functions approach also supports *incremental* and *layered* verification. When proving VC1, we are verifying the writeback stage of the pipeline against its specification `C_EX_WB`. When proving VC2, we are verifying one more stage of the pipeline, and so on. This makes it easier to locate errors. In the flushing approach, if there is a bug in the pipeline, the validity checker would produce a counterexample—a set of formulas potentially involving *all* the implementation variables—that implies the negation of the formula corresponding to the commutative diagram. Such a counterexample cannot isolate the stage in which the bug occurred.

Another advantage of using completion functions is that their definition, unlike that of flush operation, is not dependent on the latency of the pipeline. Hence our method is applicable even when the latency of the pipeline is indeterminate, which can happen if, for example, the pipeline contains data-dependent iterative loops or when the implementation machine has non-determinism. The proof that the implementation eventually reaches a flushed state can be constructed by defining a measure function that returns the number of cycles the implementation takes to flush and showing that the measure decreases after a transition from a non-flushed state.

A disadvantage of the completion functions approach is that the user must explicitly specify the definitions for these completion functions and then construct an abstraction function. In a later section, we describe a hybrid approach to reduce the manual effort involved in this process.

4 Application of Our Methodology

In this section, we explain how to apply our methodology to verify three examples: a pipelined and a superscalar pipelined implementations of a subset of the

DLX processor [7] and a processor with out-of-order execution. We describe how to specify the completion functions and construct an abstraction function, how to handle stalls, speculative fetching and out-of-order execution, and illustrate the particular decomposition and the proof strategies we used. The DLX example was verified in [3] using the flushing approach, the superscalar DLX example was verified in [2] and the processor with out-of-order execution was verified in [13]. Our verification is carried out in PVS [11]. The detailed implementation, specification as well as the proofs for all these examples can be found at [8]. The manual effort spent on developing the proofs for the processor with out-of-order execution was less than two person weeks. The first two examples were verified while developing our methodology and took about two person months, which also included the time to learn PVS.

4.1 DLX Processor Details

The specification of this processor has four state components: the program counter pc, the register file regfile, the data memory dmem, and the instruction memory imem. The processor supports six types of instructions: load, store, unconditional jump, conditional branch, alu-immediate and a 3-register alu instruction. The ALU is modeled using an uninterpreted function. The memory system and the register file are modeled as stores with read and write operations.

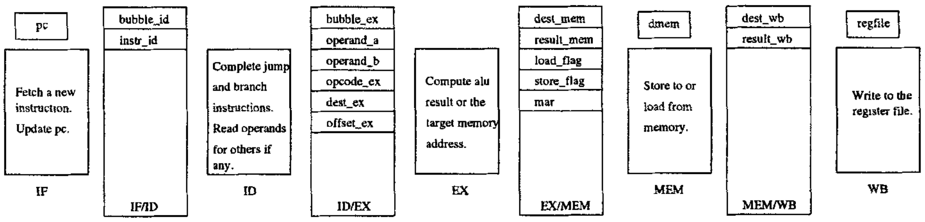
pc
Fetch a new instruction. Update pc.
IF
bubble_id
instr_id
IF/ID
Complete jump and branch instructions. Read operands for others if any.
ID
bubble_ex
operand_a
operand_b
opcode_ex
dest_ex
offset_ex
ID/EX
Compute alu result or the target memory address.
EX
dest_mem
result_mem
load_flag
store_flag
mar
EX/MEM
dmem
Store to or load from memory.
MEM
dest_wb
result_wb
MEM/WB
regfile
Write to the register file.
WB

Fig. 3. Pipelined implementation

The implementation uses a five stage pipeline as shown in Figure 3. There are four sets of pipeline registers holding information about the partially executed instructions in 15 pipeline registers. The intended functionality of each of the stages is also shown in the figure. The implementation uses a simple "assume not taken" prediction strategy for jump and branch instructions. Consequently, if a jump or branch is indeed taken (br_taken signal is asserted), then the pipeline squashes the subsequent instruction and corrects the pc. If the instruction in the IF/ID registers is dependent on a load in the ID/EX registers, then that instruction will be stalled for one cycle (st_issue signal is asserted), otherwise the instructions flow down the pipeline with every cycle. No instructions are fetched in the cycle where stall_input is asserted. The implementation provides forwarding of data to the instruction decode unit (ID stage) where the operands are read. The details of forwarding are not shown in the figure.

4.2 Specifying the Completion Functions

The processor can have four partially executed instructions at any time, one each in the four sets of pipeline registers shown. We associate a completion function with each such instruction. We need to identify how a partially executed instruction is stored in a particular set of pipeline registers—once this is done, the completion function for that unfinished instruction can be easily derived from the specification.

Consider the set IF/ID of pipeline registers. The intended functionality of the IF stage is to fetch an instruction (place it in `instr_id`) and increment the `pc`. The `bubble_id` register indicates whether the instruction is valid or not. (It might be invalid, for example, if it is being squashed due to a taken `branch`). So in order to complete the execution of this instruction, the completion function should do nothing if the instruction is not valid, otherwise it should update the `pc` with the target address if it is a `jump` or a taken `branch` instruction, update the `dmem` if it is a `store` instruction and update the `regfile` if it is a `load`, `alu-immediate` or `alu` instruction according to the semantics of the instruction. The details of how these are done can be gleaned from the specification. This function is not obtained by tracing the implementation, instead, the user directly provides the intended effect. Also note that we are not concerned with load interlock or data forwarding while specifying the completion function. We call this function `C_IF_ID`. Similarly the completion functions for the other three sets of pipeline registers—`C_ID_EX`, `C_EX_MEM` and `C_MEM_WB`—are specified.

The completion functions for the unfinished instructions in the initial sets of pipeline registers are very close to the specification and it is very easy to derive them. (For example, `C_IF_ID` is almost the same as the specification). However the completion functions for the unfinished instructions in the later sets of pipeline registers are harder to derive as the user needs to understand how the information about the instruction is stored in the various pipeline registers but the functions themselves are much simpler.

4.3 The Decomposition and the Proof Details

Since the instructions flow down the pipeline in order, the abstraction function is defined as a simple composition of these completion functions as :

```
    projection(C_IF_ID(C_ID_EX(C_EX_MEM(C_MEM_WB(impl_state)))))
```

The synchronization function returns zero if either `st_issue` or `stall_input` or `br_taken` is true, otherwise it return one.

The Decomposition. The decomposition we used for `regfile` for this example is shown in Figure 4. The justification for the first three verification conditions is similar as in Section 3. There are two verification conditions corresponding to the instruction in the IF/ID registers. If `st_issue` is true, then that instruction is not issued, so `C_IF_ID` ought not to be applied in the upper path in the commutative diagram. VC4_r requires us to prove this under condition P1 = `st_issue`. VC5_r is for the case when the instruction is issued, so it should be proved under condition P2 = `NOT st_issue`. Observe that `st_issue` also appears as a disjunct in the synchronization function and hence in `A_step'` too. Finally, VC6_r is the verification condition corresponding to the final commutative diagram for **`regfile`**.

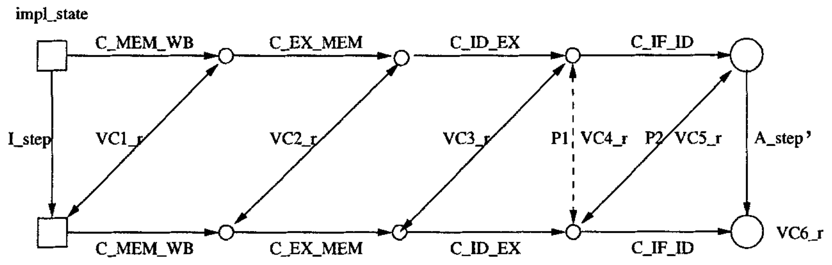

Fig. 4. The decomposition of the commutative diagram for `regfile`

The decomposition used for a particular observable depends on the pipeline stage where that observable is updated. For example, the first verification condition for `dmem` states that `dmem(C_MEM_WB(impl_state)) = dmem(impl_state)` since `dmem` is not updated in the last stage of the pipeline. Other verification conditions are exactly identical to that of **`regfile`**. Finally, the decompositions we used for `pc` and `imem` had three and two verification conditions, respectively.

The Proof. The proof is organized into three phases: first that of generating and proving certain rewrite rules, second that of proving the verification conditions and other lemmas and invariants (if needed) using these rewrite rules and third that of proving the other proof obligations mentioned in section 2.

For each register in a particular set of pipeline registers, we need a rewrite rule that states that the register is unaffected by the completion functions of the unfinished instructions ahead of it. For example, for `bubble_ex`, the rewrite rule is `bubble_ex(C_EX_MEM(C_MEM_WB(impl_state))) = bubble_ex(impl_state)`. All these rules can be generated and proved automatically by rewriting using the definitions of the completion functions. We then defined a PVS strategy that makes these rules, and the definitions and the axioms from the implementation and the specification (leaving out a few on which we do case analysis), as rewrite rules.

Now, the proof strategy for proving all the verification conditions is similar—use the PVS strategy described above to setup the rewrite rules, setup the previously proved verification conditions as rewrite rules, `expand` the outermost functions on both sides, use the PVS command **assert** to do the rewrites and simplifications by decision procedures, then perform case analysis with the PVS strategy **(apply (then* (repeat (lift-if)) (bddsimp) (ground)))**. Some verification conditions (like VC4_r) needed the outermost function to be expanded on only one side and some were slightly more involved (like VC6_r) needing case analysis on the various terms introduced by expanding **A_step'**.

The proof above needed a lemma expressing the correctness of the feedback logic. With completion functions, we could state this succinctly as follows: the value read in the ID stage by the feedback logic (when the instruction in the IF/ID registers is valid and not stalled) is the same as the value read from **regfile** after the three instructions ahead of it are completed. Its proof was done essentially with case analysis using the PVS strategy shown in the previous paragraph. We also needed an invariant on the reachable states in this example and the proof that it was closed under **I_step** was trivial.

Finally we prove that the implementation machine eventually goes to a flushed state if it is stalled sufficiently long and then check in that flushed state **fs**, **ABS(fs) = projection(fs)**. For this example, this proof was done by observing that **bubble_id** will be true after two stall transitions (hence no instruction in the IF/ID registers) and that this "no-instruction"-ness propagates down the pipeline with every stall transition. Also, we prove that the synchronization function does not always return zero which was straightforward.

4.4 Application to Superscalar DLX Processor

The superscalar DLX processor [2] is a dual issue version of the DLX processor. Both the pipelines have similar structure as Figure 3 except that the second pipeline only executes **alu-immediate** and **alu** instructions. In addition, the processor has one instruction buffer location.

Specifying the completion functions for the various unfinished instructions was similar to the DLX example. A main difference was how the completion functions of the unfinished instructions in the IF/ID registers and the instruction buffer (say the instructions are **i, j, k** and completion functions are **C_i**, **C_j** and **C_k** respectively) are composed to handle the speculative fetching of instructions. These unfinished instructions could be potential branches since the branch instructions are executed in the ID stage of the first pipeline. So while constructing the abstraction function, we compose **C_j** (with **C_i(...rest of the completion functions in order...)**) only if instruction **i** is not a taken branch and then compose **C_k** only if instruction **j** is not a taken branch too. We used a similar idea in constructing the synchronization function. The specification machine would not execute any new instructions if any of the instructions **i, j, k** mentioned above is a taken branch. It is very easy and natural to express these conditions using completion functions since we are not concerned with when exactly the branches are taken in the implementation machine. However, in the pure flushing approach, even the synchronization function will have to be much more complicated—having to cycle the implementation machine for many cycles [2].

Another difference between the two processors was the complex issue logic here which could issue zero to two instructions per cycle. We had eight verification conditions on how different instructions get issued or stalled/move around. The proofs of all the verification conditions again used very similar strategies. The synchronization function had many more cases in this example and the previously proved verification conditions were used many times over.

4.5 Application to a Processor with Out-of-order Execution

We have applied our approach to an out-of-order execution processor that was verified in [13]. This processor has three execution units—a multiplier, an adder and a load/store unit—sharing the write-back stage. This situation represents a structural hazard which is resolved by the issue logic by not issuing instructions such that two may simultaneously be in the write-back stage. We prove an invariant on the issue logic that this hazard is resolved properly and then build the proof of the commutative diagram in the various cases. Formulating the verification conditions in these various cases was similar as in the earlier examples. The interesting case is the following scenario of out-of-order completion. An `add` instruction takes one cycle in the execution unit while a `mult` takes three cycles. So, an `add` instruction, issed after a `mult` instruction, can complete before it. However, the processor would issue such an `add` instruction only if its destination register is different from that of the `mult` instruction issued earlier. We use this fact to reorder the completion functions of the `add` and the `mult` instructions into the the order used by the abstraction function.

In [13], Sawada and Hunt construct an intermediate abstraction of the implementation machine using a table that represents the (infinite) trace of all executed instructions up to the present time. They achieve incrementality by postulating and proving individually a large set of invariant properties about this intermediate representation, from which they derive the final correctness proof. The main difference of our approach is that the incremental nature of the proof in our case arises from the way we construct our abstraction function and the decomposition of the proof of the commutative diagram that it leads to. This decomposition is to a large extent independent of the processor design. Our approach also has the advantage that the amount of information the user needs to specify is significantly less than their method. For example, we require just a few simple invariants on the reachable states and do not need to construct an explicit intermediate abstraction of the implementation machine.

4.6 Hybrid Approach to Reduce the Manual Effort

In some cases, it is possible to *derive* the definitions of some of the completion functions automatically from the implementation to reduce the manual effort. We illustrate this on the DLX example.

The implementation machine is provided in the form of a typical transition function giving the "new" value for each state component. Since the implementation modifies the `regfile` in the writeback stage, we take `C_MEM_WB` to be

new_regfile which is a function of **dest_wb** and **result_wb**. To determine how **C_EX_MEM** updates the register file from **C_MEM_WB**, we perform a step of symbolic simulation of the non-observables in the definition of **C_MEM_WB**, that is, replace **dest_wb** and **result_wb** in its definition with their "new-" counterparts. Since the MEM stage updates **dmem**, **C_EX_MEM** will have another component modifying **dmem** which we simply take as **new_dmem**. Similarly we derive **C_ID_EX** from **C_EX_MEM** through symbolic simulation. For the IF/ID registers, this gets complicated on two counts: the instruction there could get stalled due to a load interlock, and the forwarding logic that appears in the ID stage. So we let the user specify this function directly. We have done a complete proof using these completion functions. The details of the proof are similar. An important difference here is that the invariant that was needed earlier was eliminated.

While reducing the manual effort, this way of deriving the completion functions from the implementation has the disadvantage that we are verifying the implementation against itself. This contradicts our view of these as *desired* specifications and negates our goal of incremental verification. To combine the advantages of both, we could use a hybrid approach where we use explicitly provided and symbolically generated completion functions in combination. For example, we could derive it for the last stage, specify it for the penultimate stage and then derive it for the stage before that (from the specification for the penultimate stage) and so on.

5 Conclusions

We have presented a systematic approach to modularize and decompose the proof of correctness of pipelined microprocessors and shown its generality by applying it to three different processors. The methodology relies on the user expressing the cumulative effect of flushing in terms of a set of completion functions, one per unfinished instruction. This method results in a natural decomposition of the proof based on the individual stages of the pipeline and allows the verification to proceed incrementally overcoming the term-size and case explosion problem of the flushing approach. While this method increases the manual effort on the part of the user, we found the knowledge required in specifying the completion functions, constructing the abstraction function and formulating the verification conditions is close to the designer's intuition about how the pipeline works.

One of our future plans is to build a system that uses PVS or a part of it as a back-end to support the methodology presented. Besides automating parts of the methodology, this system would help the user interactively apply the rest of the process. We would also like to see how our approach can be extended to verify more complex pipeline control that uses reorder buffers or other out-of-order completion techniques. Other plans include testing the efficacy of our approach for verifying pipelines with data dependent iterative loops and asynchronous memory interface.

Acknowledgements We would like to thank John Rushby and David Cyrluk for their feedback on earlier drafts of this paper.

References

1. Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Srivas and Camilleri [14], pages 187–201.
2. J. R. Burch. Techniques for verifying superscalar microprocessors. In *Design Automation Conference, DAC '96*, June 1996.
3. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In David Dill, editor, *Computer-Aided Verification, CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80, Stanford, CA, June 1994. Springer-Verlag.
4. 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.
5. David Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.
6. David Cyrluk. Inverting the abstraction mapping: A methodology for hardware verification. In Srivas and Camilleri [14], pages 172–186.
7. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.
8. Ravi Hosabettu. PVS specification and proofs of the DLX, superscalar DLX examples and the processor with out-of-order execution, 1998. Available at http://www.cs.utah.edu/~hosabett/pvs/processor.html.
9. R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *International Conference on Computer Aided Design, ICCAD '95*, 1995.
10. Jeremy Levitt and Kunle Olukotun. A scalable formal verification methodology for pipelined microprocessors. In *Design Automation Conference, DAC '96*, June 1996.
11. 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.
12. Seungjoon Park and David L. Dill. Protocol verification by aggregation of distributed actions. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 300–310, New Brunswick, NJ, July/August 1996. Springer-Verlag.
13. J. Sawada and W. A. Hunt, Jr. Trace table based approach for pipelined microprocessor verification. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 364–375, Haifa, Israel, June 1997. Springer-Verlag.
14. Mandayam Srivas and Albert Camilleri, editors. *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, Palo Alto, CA, November 1996. Springer-Verlag.
15. Mandayam K. Srivas and Steven P. Miller. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Formal Methods in Systems Design*, 8(2):153–188, March 1996.