Decomposing the Proof of Correctness of Pipelined Microprocessors

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

[1]Department of Computer Science
University of Utah
Salt Lake City, UT 84112

[2]Computer Science Laboratory
SRI International
Menlo Park, CA 94025

Contact email: hosabett@cs.utah.edu

January 12, 1998

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 specify the effect (on the observables) of completing the instruction. In addition to avoiding term-size and case explosion as could happen for deep and complex pipelines during *flushing* and helping localize errors, our method can also handle stages with iterative loops. The technique is illustrated on pipelined- as well as a superscalar pipelined implementations of a subset of the DLX architecture.

Keywords: Processor verification, Decomposition, Incremental verification
Category: A

1 Introduction

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 commute diagram. Second, they use symbolic simulation to derive logical expressions corresponding to the two paths in the commute diagram which will be then 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 [CRSS94, SM96]. Some of the approaches to pipelined processor verification rely on the user providing the definition for the abstraction function. Burch and Dill in [BD94] observed that the 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 [JDB95, BDL96] to effectively automate the verification of pipelined implementations of several processors.

The pure flushing approach has the drawback of generating an impractically large abstraction function for deeper pipelines. Also, the number of examined cases explodes as the control part becomes complicated. To overcome this drawback, Burch [Bur96] decomposed the verification problem into three subproblems and suggested an alternative method for constructing the abstraction function. This method required the user to add some extra control inputs to the implementation and set them appropriately while constructing the abstraction function. Along with a validity checker which needed the user to help with many manually derived case splits, he used these techniques in superscalar processor verification. However, despite the manual effort involved, the reduction obtained in the expression size and the number of cases explored as well as how the method will scale is not clear.

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. Thus, in addition to actually gaining from designers' insights, verification based on the completion function method has a number of other advantages. It results in a natural decomposition of proofs. Proofs builds 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. The method is not explicitly targeted towards any single aspect of processor design such as control, and can naturally handle loops in pipeline stages.

1.1 Related work

Cyrluk has developed a technique called 'Inverting the abstraction mapping' [Cyr96] for guiding theorem provers during processor verification. In addition to not decomposing proofs in our sense, this technique also suffers from large term sizes. Park and Dill have used the idea of aggregation functions in distributed cache coherence protocol verification [PD96]. The completion functions are similar to aggregation functions but our goal is the decomposition of the proof we can achieve using them. Additional comparisons with past work are made in subsequent sections.

2 Correctness Criteria for Processor Verification

The completion functions approach aims to realize the correctness criterion expressed in Figure 1(a) (used in [SH97]), in a manner that proofs based on it are modular and layered as pointed out earlier. Figure 1(a) expresses that `n` implementation transitions which start and end with flushed states correspond to `m` transitions in the specification machine where `m` is the number of instructions executed in the specification machine. `I_step` is the implementation transition function and `A_step` is the specification transition function. `projection` would extract only those implementation state components visible to the specification i.e. the observables. This criterion is preferred 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. One way to adapt this correctness criterion into an inductive argument would be to first show that the processor meets the criterion in Figure 1(b), and then check that the abstraction function `ABS` satisfies the condition that in a flushed state `fs`, `ABS(fs) = projection(fs)`. One also needs to prove that the implementation machine will eventually reach a flushed state if no more instructions are inserted into the machine. This is to make sure that the correctness criterion in Figure 1(a) is not vacuous.

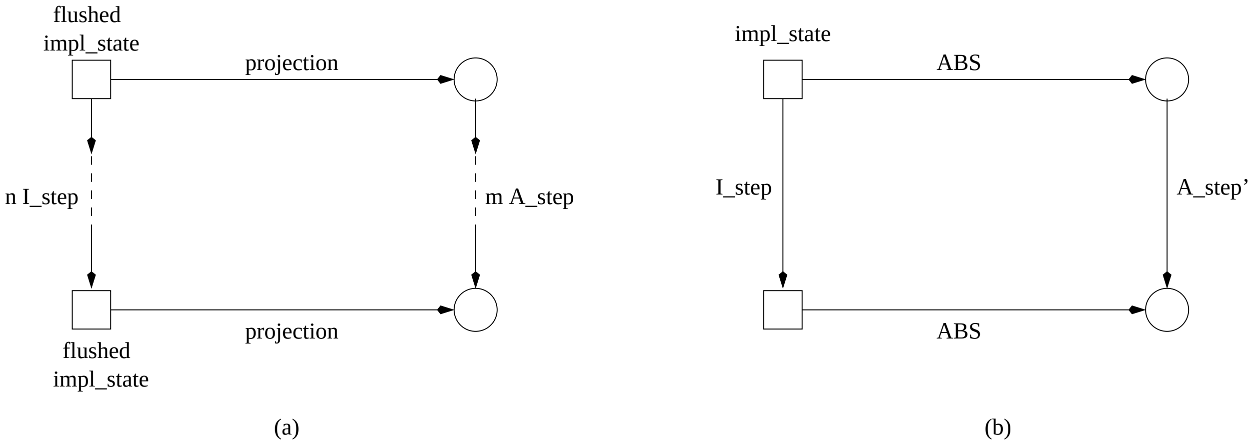

Figure 1: Pipelined microprocessor correctness criteria

Intuitively, Figure 1(b) says that if the implementation and the specification machines start in a corresponding pair of states, then after executing a transition, their new states correspond. `impl_state` is an arbitrary reachable state of the implementation machine. Figure 1(b) uses a modified transition function `A_step'` instead of `A_step` since certain implementation transitions might correspond to executing zero, or more than one instructions in the specification machine. The case of zero instruction can arise if, *e.g.*, the implementation machine stalls due to a load interlock. The case of more than one instruction can arise if, *e.g.*, the the implementation machine has multiple pipelines. The number of instructions executed by the specification machine is provided by a function on implementation states (called the synchronization function). One of the crucial proof obligations is to show that this function does not always return zero.

The most difficult task here is to define an appropriate abstraction function and to prove that the Figure 1(b) commutes. One way to define an abstraction function [BD94] is to flush the pipeline so that all the unfinished instructions complete, and update the observables, and then apply a projection. Since most machines allow for stalling the pipeline, i.e., advancing the implementation machine without fetching a new instruction, flushing can be performed by a sequence of stall transitions of the implementation machine. The number of stall transitions required depends on the depth of the pipeline, stall cycles due to interlocks etc. This would generate the following verification condition for proving that Figure 1(b) commutes (where `flush` is as discussed before):

```
Flush_VC: A_step(projection(flush(impl_state))) = projection(flush(I_step(impl_state)))
```

It is practical to prove this verification condition only for simple and shallow pipelines. For superscalar processors with multiple pipelines and complex control logic, the logical expressions generated are too large to manage and check equivalence on. Another drawback is that the number of stall transitions to flush the pipeline should be known, a priori. This, even if finite, may be indeterminate 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.

3 The Completion Functions Approach

The completion functions approach is also based on using an abstraction function corresponding to flushing the entire pipeline. However, this function is *not* derived via flushing in our basic approach[1]. Rather, we construct the abstraction function as a composition of a sequence of completion functions which, as said earlier, specifies the *desired effect* (on the observables) of completing each unfinished instruction. These completion functions must also leave 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 *e.g.*, 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. As we illustrate later, this definition of the abstraction function leads to a very natural decomposition of the proof of the commute diagram and supports incremental verification. Any mistakes, either in specifying the completion functions or 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. (This is unrealistically simple, but we explain how to handle these artifacts in subsequent sections). There can be three unfinished instructions in this pipeline at any time, 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) would state how the different values stored in that set of registers (ID/EX in this example) are combined to complete that instruction. In our example, the completion functions are `C_EX_WB`, `C_ID_EX` and `C_IF_ID`. 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)))
```

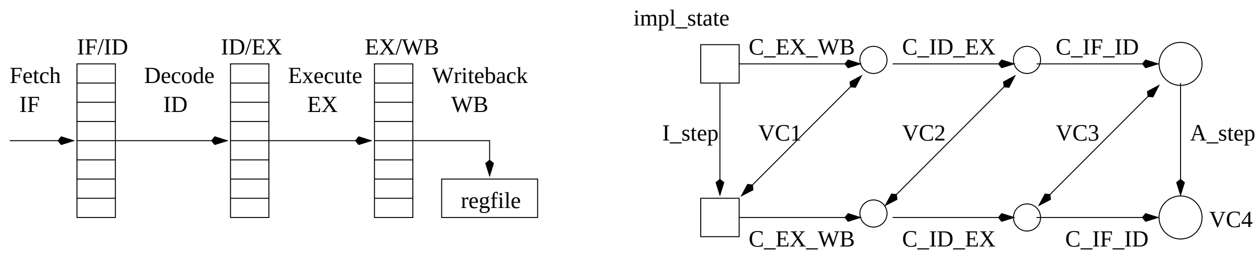

Figure 2: A simple four stage pipeline and decomposition of the proof under completion functions

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

[1]Later we discuss a hybrid scheme extension

```
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 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 set EX/WB of pipeline 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 set EX/WB of pipeline registers. `C_EX_WB` can now take over and complete this instruction. This would 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 similar. 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* which involves expanding the outermost function on the both sides of the equation and using the previously proved verification condition (if any) as a rewrite rule to simplify the expressions, followed by the necessary case analysis, as well as reasoning about the terms introduced by function expansions. Since we expand only the topmost functions on both sides, and because we use the previously proved verification condition, the sizes of the expressions produced during the proof and the required case analysis are kept in check.

As mentioned earlier, 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 is easier to locate errors. In [BD94], 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 `Flush_VC`. Such an output is not helpful in pinpointing the bug.

Another important advantage of the completion functions method is that it is applicable even when the number of stall transitions to flush the pipeline is indeterminate, which can happen if, *e.g.*, the pipeline contains data dependent iterative loops. The completion functions, which state the desired effect of completing an unfinished instruction, help us express the effect of flushing directly. The proof that the implementation eventually goes to a flushed state can be done by using a measure function which returns the number of cycles the implementation takes to flush (this will be a data dependent expression, not a constant) and showing that either the measure function decreases after every cycle or the implementation machine is flushed.

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 to DLX and Superscalar DLX Processors

In this section, we explain how to apply our methodology to verify two examples - a pipelined and a superscalar pipelined implementation of a subset of the DLX processor [HP90]. We describe how to specify the completion functions and construct an abstraction function, how to handle stalls, speculative fetching and certain hazards, and illustrate the particular decomposition and the proof strategies that we used. These are the same examples that were verified by Burch and Dill using the flushing approach in [BD94] and by Burch using his techniques in [Bur96] respectively. Our verification is carried out in 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`. There are six types of instructions supported: `load`, `store`, unconditional `jump`, conditional `branch`, `alu-immediate` and 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. The semantics of `read` and `write` operations are provided using the following two axioms: `addr1 = addr2 IMPLIES read(write(store,addr1,val1),addr2) = val1` and `addr1 /= addr2 IMPLIES read(write(store,addr1,val1),addr2) = read(store,addr2)`. The specification is provided in the form of a transition function `A_step`.

The implementation is 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 diagram. 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 following a `load` is dependent on it (`st_issue` signal is asserted), then that instruction will be stalled for a cycle in the set IF/ID of pipeline registers, otherwise they 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 diagram. The implementation is also provided in the form of a transition function `I_step`. The detailed implementation, specification as well as the proofs can be found at [Hos98].

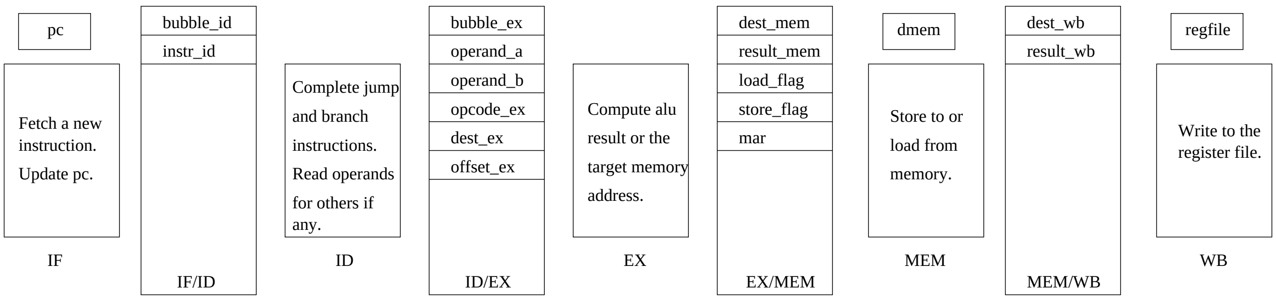

Figure 3: Pipelined implementation

4.2 Specifying the completion functions

There can be four partially executed instructions in this processor 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 is in 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.

Consider the set ID/EX of pipeline registers. The ID stage completes the execution of jump and branch instructions, so this instruction would affect only dmem and regfile. The bubble_ex indicates whether the instruction is valid or not, operand_a and operand_b are the two operands read by the ID stage, opcode_ex and dest_ex determine the opcode and the destination register of the instruction and offset_ex is used to calculate the memory address for load and store instructions. The completion function should state how these information can be combined to complete the instruction, which again can be gleaned from the specification. We call this function C_ID_EX. Similarly the completion functions for the other two sets of pipeline registers - 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 for the unfinished instructions in the later sets of pipeline registers, it is more involved to derive them as the user needs to understand how the information about unfinished instructions are stored in the various pipeline registers but the functions themselves are much simpler. Also the completion functions are independent of how the various stages are implemented and just depend on their functionality.

4.3 The decomposition and the proof details

Since the instructions flow down the pipeline in order, the abstraction function is defined the composition of these completion functions followed by projection as shown below:

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

The synchronization function, for this example, returns zero if there is a load interlock (st_issue is true) or stall_input is asserted or jump/branch is taken (br_taken is true) otherwise it returns one. The modified specification transition function is A_step'. The proof that this function is not always zero was straightforward and we skip the details here. This is also needed in the approach of [BD94].

4.3.1 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 set IF/ID of pipeline registers. If st_issue is true, then that instruction is not issued, so C_ID_EX ought to have no effect in the lower path in the commute 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. VC6_r is the verification condition corresponding to the final commute diagram for regfile.

The decomposition for dmem is similar except that the first verification condition VC1_d is slightly different. Since dmem is not updated in the last stage, VC1_d for dmem states that dmem is not affected by C_MEM_WB i.e. dmem(C_MEM_WB(impl_state)) = dmem(impl_state). The rest of the verification conditions are exactly identical to that of regfile.

The commute diagram for pc was decomposed into only three verification conditions. We first one, VC1_p, stated that pc(C_ID_EX(C_EX_MEM(C_MEM_WB(impl_state)))) = pc(impl_state) since completing the instructions in the last three sets of pipeline registers will not affect the pc. In addition, completing the instruction in set IF/ID of pipeline registers will not affect the pc too, if that instruction is not stalled

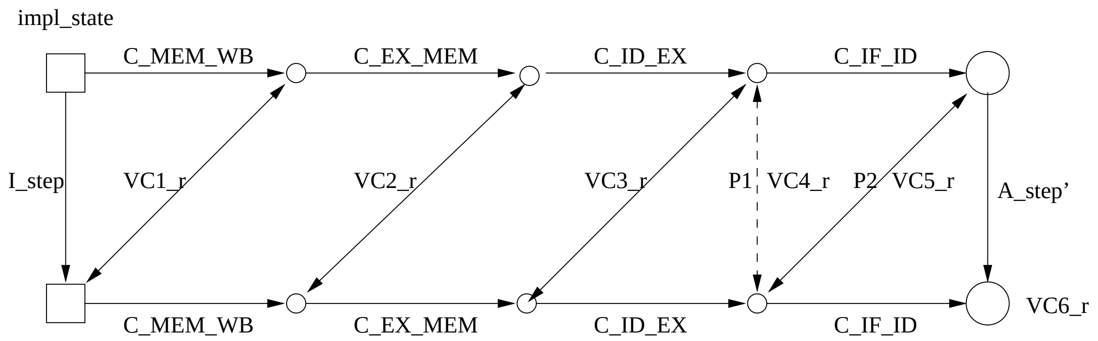

Figure 4: The decomposition of the commute diagram for `regfile`

and is not a `jump`/taken `branch`. This is captured by VC2_p. The third one, VC3_p, was the verification condition corresponding to the final commute diagram for `pc`.

The decomposition we used for `imem` had two verification conditions: VC1_i which stated that completing the four instructions in the pipeline has no effect on `imem` and the second one, VC2_i was corresponding to the final commute diagram for `imem`.

4.3.2 The proof

We need a rewrite rule for each register of a particular set of pipeline registers that states that it 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. We then defined a strategy which would setup these, and the definitions and the axioms from the implementation and the specification as rewrite rules. We avoid setting up as rewrite rules those definitions on which we do case analysis - `st_issue` and `br_taken` and those corresponding to the feedback logic.

The correctness of the feedback logic is captured succinctly in the form of following two lemmas, one each for the two operands that it reads. If there is a valid instruction in set IF/ID of pipeline registers and it is not stalled, then the value read in the ID stage by the feedback logic is the same as the value read from `regfile` after the three instructions ahead of it are completed. Their proofs are done by using the strategy above to setup all the rewrite rules, setting up the definitions in the lemmas being proved as rewrite rules, followed by an `assert` to do the rewrites and simplifications, followed by `(apply (then* (repeat (lift-if)) (bddsimp) (ground)))` to do the case analysis.

The proof strategy for proving all the verification conditions of `regfile` and `dmem` is similar - use the strategy described above to setup the rewrite rules, set up the previously proved verification conditions and the lemmas about feedback logic as rewrite rules, `expand` the outermost function on both sides, `assert` to do the rewrites and simplifications, then do case analysis with `(apply (then* (repeat (lift-if)) (bddsimp) (ground)))`. Minor differences were that some finished without the need for case analysis (like VC2_r and VC2_d) and some needed the outermost function to be expanded on only one of the sides (like VC4_r and VC4_d). VC6_r and VC6_d were slightly more involved in that the various cases introduced by expanding `A_step'` were considered in the following order - `st_issue, stall_input, br_taken` - followed by a similar strategy as described before.

The proofs of the verification conditions for `pc` were again similar except that we do additional case analysis after expanding `br_taken` condition. Finally, the proofs of verification conditions for `imem` were trivial since the instruction memory does not change.

We needed an invariant in this example: that `dest_ex` is `zero_reg` whenever `bubble_ex` is true or `opcode_ex` is a `store` or a `jump` or a `branch` instruction. Making `dest_ex` equal to `zero_reg` was to ensure that the `regfile` was not updated under these conditions. The proof that the invariant is closed under `I_step` was however trivial.

We make two observations here. The proof of a particular verification condition, say for `regfile`, may use the previous verification conditions of all other specification state components, hence these need to be proved in that order. The particular order in which we did the proof was VC1_r, VC1_d, VC2_r, VC2_d, VC3_r, VC3_d, the two lemmas for feedback logic, VC4_r, VC4_d, VC5_r, VC5_d, VC1_i, VC1_p, VC2_p, VC6_r, VC6_d, VC3_p and VC2_i. The second observation is that this is the particular decomposition that we chose. We could have avoided proving, say VC4_r, and proved that goal when it arises within, say VC6_r, if the prover can handle the term sizes.

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 set IF/ID of pipeline registers) and that this 'no-instruction'-ness propagates down the pipeline with every stall transition.

4.4 Superscalar DLX processor

The superscalar DLX processor 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, there is one instruction buffer location.

Specifying the completion functions for the various unfinished instructions was similar. A main difference was how the completion functions of the unfinished instructions in the sets IF/ID of pipeline 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 too. 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, if using the pure flushing approach, even the synchronization function will have to be much more complicated having to cycle the implementation machine for many cycles [Bur96].

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. (This again is the particular decomposition that we chose, we can reduce this by choosing a coarser decomposition). The complete PVS specification and proofs can be found at [Hos98]. 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 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 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`. This is a function of `dest_wb` and `result_wb`. To determine how `C_EX_MEM` updates the register file, we perform a step of symbolic simulation of the non-observables i.e. replace `dest_wb` and `result_wb` in above function 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 set IF/ID of pipeline 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 this eliminated the invariant that was needed earlier.*

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. In the example above, a bug in the writeback stage would go undetected and appear in the completion functions that are being built up. (In fact, VC1_r for `regfile` is true by the construction of `C_MEM_WB` and hence need not be proved - we believe, we can formalize this under suitable assumptions on the implementation). All bugs will eventually be caught however, since the final commute diagram uses the 'correct' specification provided by the user instead of being generated from the implementation. 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 it (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. This relied on the user expressing the cumulative effect of flushing in terms of a set of completion functions, one per unfinished instruction. This resulted in a natural decomposition of the proof and allowed the verification to proceed incrementally. While this method increased the manual effort on the part of the user, we found specifying the completion functions and constructing the abstraction function was quite easy and believe that a typical designer would have an understanding of these. We also believe that our approach can verify deeper and complex pipelines than is possible with other automated methods.

Our future plan is to see how our approach can be applied, or can be adapted, to verify more complex pipeline control that use out-of-order completion of instructions. Our initial attempts at verifying such a processor appear encouraging. The particular processor we are attempting to verify allows out-of-order completion of instructions but has a complex issue logic that allows such a possibility only if that instruction does not cause any *WAW* hazards. The crucial idea here is that we can reorder the completion functions of the unfinished instructions to match the program order (the order used by the abstraction function) using this property of the issue logic. 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 for very useful feedback on the first draft of this paper.

References

[BD94] 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.

[BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Srivas and Camilleri [SC96], pages 187–201.

[Bur96] J. R. Burch. Techniques for verifying superscalar microprocessors. In *Design Automation Conference, DAC '96*, June 1996.

[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.

[Cyr96] David Cyrluk. Inverting the abstraction mapping: A methodology for hardware verification. In Srivas and Camilleri [SC96], pages 172–186.

[Hos98] Ravi Hosabettu. PVS specification and proofs of DLX and superscalar DLX examples, 1998. Available at http://www.cs.utah.edu/~hosabett/pvs/dlx.html.

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

[JDB95] 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.

[PD96] 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.

[SC96] 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.

[SH97] 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.

[SM96] 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.