SYSTEMATIC VERIFICATION OF PIPELINED MICROPROCESSORS

by

Ravi Mohan Hosabettu

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science
The University of Utah
August 2000
THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Ravi Mohan Hosabettu

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair:  Ganesh C. Gopalakrishnan

Erik L. Brunvand

Alan L. Davis

Chris J. Myers

Mandayam K. Srivas
THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Ravi Mohan Hosabettu in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

______________________________
Date

Ganesh C. Gopalakrishnan
Chair, Supervisory Committee

Approved for the Major Department

______________________________
Thomas C. Henderson
Chair/Dean

Approved for the Graduate Council

______________________________
David S. Chapman
Dean of The Graduate School
ABSTRACT

This dissertation addresses the problem of formally verifying the correctness of pipelined microprocessors at the micro-architectural level of abstraction. Contemporary processor designs are highly complex, employing sophisticated performance enhancing techniques such as superscalar pipelining, out-of-order execution, branch prediction and speculative execution. Traditional simulation based validation methods do not guarantee that they uncover all the complex design bugs, and hence there is a need for formally verifying the correctness of these designs.

We propose a systematic approach called the Completion Functions 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 programmer visible state components of completing the instruction. This construction of the abstraction function leads to a very natural decomposition of the proof into proving a series of verification conditions. The approach prescribes a systematic way to generate these verification conditions which can then be discharged with a high degree of automation using techniques based on decision procedures and rewriting. The approach does not involve the construction of an explicit intermediate abstraction, supports incremental verification facilitating debugging and error localization, and is applicable uniformly on a wide variety of pipelined processor designs.

The methodology is implemented in PVS (a theorem prover from SRI International), which supports many decision procedures and rewriting, and has been applied to many example processor designs with reasonable manual effort. The most involved design verified is an example out-of-order execution processor with a reorder buffer, a store buffer, branch prediction, speculative execution and exceptions. The verification was completed in 34 person days, which we believe, is a modest investment in return for the significant benefits of formal verification.
To my parents
CONTENTS

ABSTRACT .................................................................................. iv

LIST OF FIGURES ......................................................................... ix

LIST OF TABLES ........................................................................... x

ACKNOWLEDGEMENTS ................................................................. xi

CHAPTERS

1. INTRODUCTION ................................................................. 1
   1.1 Formal Verification for Functional Validation of Pipelined Processors .... 1
   1.2 Requirements of the Methodology ............................................ 2
   1.3 Contributions of the Dissertation ............................................ 4
   1.4 Related Work ........................................................................ 6
   1.5 Organization of the Dissertation ............................................ 9

2. MODELING PIPELINED PROCESSOR CORRECTNESS ......... 10
   2.1 Machine Descriptions .......................................................... 10
   2.2 Correctness Criterion .......................................................... 12

3. THE COMPLETION FUNCTIONS APPROACH .................. 15
   3.1 Compositional Construction of the Abstraction Function ......... 16
      3.1.1 Finding the unfinished instructions and their program order .... 16
      3.1.2 Specifying the completion functions ............................... 17
      3.1.3 Constructing the abstraction function ............................ 20
      3.1.4 Defining the synchronization function ......................... 20
   3.2 Decomposing the Proof of the Commutativity Obligation ....... 21
      3.2.1 Illustration on a simple example .................................... 21
      3.2.2 Generating and proving the verification conditions .......... 24
      3.2.3 Correctness of the source operands of the instructions ...... 28
      3.2.4 Proving the commutativity obligation ............................ 29
      3.2.5 Invariant properties ..................................................... 29

4. PROCESSORS WITH IN-ORDER EXECUTION AND IN-ORDER COMPLETION ........................................ 30
   4.1 Application to EX1.1 .......................................................... 30
      4.1.1 Details of EX1.1 ............................................................ 30
      4.1.2 Compositional construction of the abstraction function ...... 32
         4.1.2.1 Unfinished instructions and the program order .......... 33
         4.1.2.2 Specifying the completion functions ....................... 33
         4.1.2.3 Constructing the abstraction function ..................... 35
4.1.2.4 Defining the synchronization function ..................... 36
4.1.3 Formulating the verification conditions .......................... 37
4.1.4 Proof details ................................................. 42
  4.1.4.1 An example proof: proof of $Vc2_{rf}$ ...................... 42
  4.1.4.2 Rewrite rules and proof strategies ........................ 44
  4.1.4.3 Proving the verification conditions ....................... 45
  4.1.4.4 Main proof obligations .................................. 47
4.1.5 Some observations ........................................... 48
4.1.6 Reducing the manual effort in specifying the completion functions .. 48
4.2 Application to EX1.2 ........................................... 50
  4.2.1 Processor details .......................................... 50
  4.2.2 Completion functions and the abstraction function .......... 51
  4.2.3 Decomposition and the proof details ......................... 54

5. PROCESSORS WITH OUT-OF-ORDER EXECUTION AND
IN-ORDER COMPLETION ........................................... 61
  5.1 Details of EX2.2 .............................................. 61
  5.2 Compositional Construction of the Abstraction Function .......... 66
    5.2.1 Unfinished instructions and the program order ............. 66
    5.2.2 Specifying the completion functions ....................... 66
    5.2.3 Constructing the abstraction function ..................... 70
    5.2.4 Defining the synchronization function .................... 71
  5.3 Decomposing the Proof ....................................... 72
    5.3.1 Formulating the different verification conditions .......... 73
      5.3.1.1 Verification conditions for case 1 .................... 73
      5.3.1.2 Verification conditions for case 2(a) ................. 74
      5.3.1.3 Verification conditions for case 2(b) ................. 78
      5.3.1.4 Verification conditions for case 2(c) ................. 80
      5.3.1.5 Verification conditions for case 2(d) ................. 81
      5.3.1.6 Verification conditions for case 2(e) ................. 82
    5.3.2 Invariants and other properties needed .................... 83
      5.3.2.1 Exclusiveness and exhaustiveness of instruction phases .... 83
      5.3.2.2 Instruction phase properties .......................... 83
      5.3.2.3 Program order property in the LS Unit ................ 85
      5.3.2.4 Properties about the committed instructions in the store buffer ... 86
      5.3.2.5 Register translation table properties .................. 86
      5.3.2.6 Properties about the feedback logic .................... 87
      5.3.2.7 Properties about the latest instructions in the store buffer ... 88
      5.3.2.8 Miscellaneous invariant properties .................... 88
      5.3.2.9 Other properties ..................................... 89
    5.3.3 Discharging the different proof obligations ............... 90
      5.3.3.1 Rewrite rules and proof strategies .................... 90
      5.3.3.2 Proving the verification conditions ................. 91
      5.3.3.3 Proving the invariants and the other properties ....... 97
      5.3.3.4 Proving the commutativity obligation ................ 97

vii
6. PROCESSORS WITH OUT-OF-ORDER COMPLETION ......... 99
   6.1 Application to EX3.1 .................................................. 100
      6.1.1 Processor details .............................................. 100
      6.1.2 Constructing the abstraction function .................... 100
      6.1.3 Proof decomposition ........................................... 102
   6.2 Application to EX3.2 .................................................. 106
      6.2.1 Processor details .............................................. 106
      6.2.2 Why is the basic completion functions approach hard to apply? ... 108
      6.2.3 Constructing the abstraction function .................... 108
      6.2.4 Proof decomposition ........................................... 110
         6.2.4.1 A key verification condition and its proof .................. 110
         6.2.4.2 Proving the commutativity obligation .................. 113
         6.2.4.3 Invariant properties needed .............................. 114

7. CONCLUSIONS ............................................................. 116

APPENDICES
A. BRIEF INTRODUCTION TO PVS ................................. 119
B. MODELING THE MACHINES ......................................... 124
C. PVS STRATEGIES USED IN OUR APPROACH ..................... 129
D. PVS PROOF ORGANIZATION AND AN EXAMPLE PROOF ....... 132

REFERENCES ............................................................... 142
LIST OF FIGURES

2.1 Illustration of the pipelined microprocessor correctness criterion ........ 12
3.1 Proof decomposition illustrated on a simple example ..................... 22
3.2 Depicting the instructions and their phases, before and after an implementa-
                tion transition .............................................. 25
4.1 Pipelined implementation of EX1.1 .................................. 31
4.2 Verification conditions for rf and pc in the EX1.1 example ............... 41
4.3 Dual issue implementation of the example considered earlier .......... 51
4.4 Verification conditions for rf due to IF/ID₁ and IF/ID₂ ................... 55
5.1 Block diagram model of our implementation of EX2.2 ..................... 63
5.2 Various phases an instruction can be in and transitions between them (when
          the processor is not being restarted) .................................. 76
5.3 Scenario corresponding to abs_src2_value_feedback_VC verification condition 96
6.1 Implementation model of EX3.1 ........................................ 101
6.2 Verification conditions for rf due to the first four instructions in three of
          the cases (m, 1, o) .................................................. 103
6.3 Implementation model of EX3.2 ........................................ 107
6.4 Various phases an instruction can be in and transitions between them, I:
          issued, D: dispatched, E: executed .............................. 111
D.1 Theory organization in all the examples described in this dissertation .... 133
LIST OF TABLES

4.1 Verification conditions generated for Ex1.1 .......................... 42
4.2 Verification conditions generated for Ex1.2 .......................... 60
5.1 Verification conditions generated for Ex2.2 .......................... 84
5.2 List of invariant properties in the proof of Ex2.2 ..................... 88
7.1 Examples verified and the effort needed ............................... 116
ACKNOWLEDGEMENTS

I like to thank

• my advisors Ganesh Gopalakrishnan and Mandayam Srivas for motivating me to work in the area of pipelined processor verification, and for their guidance, encouragement and patience throughout this dissertation work.

• my committee members Erik Brunvand, Alan Davis and Chris Myers for their valuable comments and suggestions.

• Utah Verifier group—Abdel Mokkedem, Ratan Nalumas, Rajnish Ghughal, Mike Jones, Annette Bunker, Ritwik Bhattacharya, Ali Sezgin and Prosenjit Chatterjee—for being a great group to work with, and for patiently listening to my talks and suggesting ideas to improve my work.

• John Rushby for sponsoring two summer internships at CSL SRI, and the researchers there for helping me out with PVS.

• My friends at the University of Utah, in particular, Anand Ranganathan, Arcot Preetham, Ajay Chitturi, Ravi Kuramkote, Sai Susarla, for making my time at the university enjoyable.

• James Johnson, for being a great companion to stay with and for all his help.

• my parents, sisters, brothers-in-law and family members for their love, support and encouragement.

• my wife, Sahana, for her patience, support and charm.

The work presented in this dissertation is an extension and modification of the work reported in the following four papers:

1. *Decomposing the Proof of Correctness of Pipelined Microprocessors* [29]
2. Proof of Correctness of a Processor with Reorder Buffer using the Completion Functions Approach [30]

3. A Proof of Correctness of a Processor Implementing Tomasulo’s Algorithm without a Reorder Buffer [27]

4. Verifying Microarchitectures that Support Speculation and Exceptions [28]
CHAPTER 1

INTRODUCTION

1.1 Formal Verification for Functional Validation of Pipelined Processors

Functional validation of modern microprocessors is a problem of enormous significance. Contemporary microprocessors are already highly complex employing multiple performance enhancing techniques such as superscalar pipelining, out-of-order execution, branch prediction and speculative execution [21, 23, 25, 33]. Out-of-order execution allows the processors to execute the instructions out of program order, respecting only the dependencies between them. To respect the in-order completion semantics, the processors use structures such as reorder buffers to store the program order of the executing instructions. Speculation allows the processors to predict the outcome of a branch instruction and continue the execution without waiting for the actual outcome to be decided. If the speculation turns out to be incorrect, the processor needs to squash all the mispredicted instructions. This high degree of concurrency due to the simultaneous execution of the multiple instructions in the pipeline leads to many complex interactions among the executing instructions. Given that there are immense architectural design challenges ahead, some of them caused by ever-growing wiring delays and the growing disparity between the processor speeds and the memory speeds, future processors will become even more complex. While traditional simulation based validation methods are often quite effective in finding design bugs, there is no guarantee that they cover all these complex scenarios and unearth all the design bugs. Another disadvantage is that these simulation methodologies are generally applicable only late in the design cycle. The design bugs that escape detection and slip into silicon or those detected late in the design cycle are either very costly or impossible to fix. A system architect attempting these complex processor designs needs a different validation methodology applicable during the early stages of the architectural design. Formal verification applied early in the design cycle provides a means of detecting all the design bugs and accomplishing functional validation.
In an article surveying the state-of-the-art in microprocessor validation in *IEEE Micro* [7], Bose, Conte and Austin argue that higher-level models such as "pre-RTL, cycle-accurate functional models" may become a necessary step in most design methodologies, and foresee the increasing use of formal techniques in the functional validation of such models.

Very early efforts to formally verify non-pipelined microprocessors were carried out by Cohn [15], and later by Hunt [32]. One of the earliest works on verifying pipelined processors was by Srivas and Bickford in the Mini-Cayuga verification effort [50]. In 1994, Burch and Dill [13] introduced an automatic approach to verify pipelined processors and were successful in verifying many simple processor models. Several other researchers have extended their work to verify more involved models and some other new approaches have been proposed (described in a later section), but the major problems such as the handling of all the architectural features discussed earlier (out-of-order execution, speculation, exceptions, etc.) remained.

What makes the pipelined processor verification so hard? The main source of complexity is in the high degree of concurrency due to the simultaneous execution of the multiple instructions which leads to many complex interactions among the executing instructions. A given instruction may be dependent on an earlier instruction to produce one of its source values, which in turn may be dependent on some other instruction. Several instructions may compete for a resource such as an execution unit and this contention needs to be resolved. Supporting speculation and exceptions require mechanisms to roll back to a state consistent with the specification machine. Formal verification requires that all these possibilities be analyzed. Since there are a very large number of cases to be considered, there is need for a methodology that prescribes a systematic way of managing this complexity of pipelined processor verification. None of the existing techniques are capable of systematically handling designs with all the architectural features discussed earlier nor are uniformly applicable to a wide variety of designs. This makes it difficult to learn these techniques and to reuse the proof infrastructure in a new processor verification effort.

### 1.2 Requirements of the Methodology

We are seeking a methodology that lets us formally verify the functional correctness of complex pipelined processor designs at the micro-architectural level of abstraction (pre-RTL, cycle accurate model with some functional abstractions [7]). A relatively popular
and automatic verification technology that has found widespread use in industry is that of model checking [14, 44]. Classical model checking takes a finite-size model of the implementation machine and verifies that it has certain desired properties expressed in a suitable logic. Verification is accomplished automatically by using algorithmic techniques to search the state space exhaustively. A disadvantage of model checking is that it is hard to incorporate uninterpreted functions. The use of uninterpreted functions is a natural way to abstract away from the uninteresting details in the implementation, for example, relating to the functional units. Their use allows the verification to be carried out independent of any concrete implementation for those uninterpreted functions. While some progress has been made in combining model checking and uninterpreted functions [5], they have not been demonstrated on large examples. Also many successful applications of model checking are largely for debugging control intensive implementation models, with data heavily abstracted, whereas in pipelined processor designs, there is a tight coupling between data and control. Theorem provers, on the other hand, are based on deductive techniques, typically have a very expressive specification language, provide different powerful abstraction mechanisms including the use of uninterpreted functions, and can handle generic implementation models. Notable examples of theorem provers include PVS [40], ACL2 [36] and HOL [20]. A disadvantage of theorem provers is that they typically require user guidance to accomplish the proof. One way to increase the automation in theorem provers is to rely on proof strategies that are centered on the use of decision procedures [39, 49] for certain theories (or their combinations) adequate for modeling hardware designs. These theories include the theory of boolean expressions with uninterpreted functions and the theory of linear arithmetic. However just as model checking suffers from a state-explosion problem, a verification strategy based on decision procedures can suffer from a “case-explosion” problem. That is, when applied naively, the sizes of the terms generated and the number of cases examined while checking the validity of the correctness assertion explodes. A methodology that prescribes a systematic way to decompose the overall correctness assertion into smaller problems that can then be handled by strategies based on decision procedures is needed.

We therefore impose the following requirements on a methodology that supports pipelined processor verification:

1. It must allow one to focus on the flow of instructions in the pipeline only, abstracting away from details such as the exact implementation of execution units, branch pre-
diction units, etc. These issues can be verified by specialized techniques developed for them and are uninteresting in our work.

2. It must allow the design to be verified in its full generality rather than permit only a fixed size instantiation of the design to be verified. This avoids the need of re-verifying the design when some parameters such as the number of registers or the number of execution units are changed.

3. It must support verification decomposition, that is, a systematic approach to break down the overall correctness criterion into smaller assertions. It must also support error localization to facilitate debugging. Proof decomposition also helps avoid global re-verification when local changes to the micro-architectural design are made.

4. It must provide a high degree of automation in discharging the different proof obligations generated by the process of proof decomposition.

5. It must be uniformly applicable to a wide class of pipelined designs. This requirement is important for several reasons: (i) A variety of pipelining styles will be in practice at any point in time. (ii) It helps amortize the effort to learn the methodology and promotes re-use of verification strategies. (iii) Variations in the micro-architectural design can be accommodated by re-parameterizing the key steps in the methodology rather than being forced to switch to another technique.

1.3 Contributions of the Dissertation

In this dissertation, we propose a systematic methodology called the Completion Functions Approach to decompose and incrementally build the proof of correctness of pipelined microprocessors, and meeting all the requirements listed in the earlier section. The central idea behind this approach is proof decomposition. This is facilitated by a compositional construction of the abstraction function used in the correctness criterion to relate the implementation and the specification machines, using a collection of completion functions. There is one completion function corresponding to every unfinished instruction flowing in the pipeline. A completion function specifies the effect of completing a given unfinished instruction in an atomic fashion on the programmer visible state components. Given such a construction of the abstraction function in terms of completion functions, our verification methodology prescribes a way of organizing the verification into proving
a series of verification conditions. Our methodology has the following features:

- The specification of the completion functions is a very natural activity amounting to expressing the knowledge that the designer already possesses about the design. Mistakes (if any) made in specifying completion functions can never lead to false positive results.

- The verification conditions and most of the supporting lemmas needed to accomplish the proof can be generated systematically. They can also be discharged with a high degree of automation using strategies based on decision procedures and rewriting.

- Verification proceeds incrementally, debugging the pipeline stages one at a time, thus facilitating error localization.

- The methodology involves reasoning about the implementation machine directly. It avoids the construction of an explicit intermediate abstraction as well as the large amount of manual effort required in the process.

- The methodology is applicable primarily for in-order completion machines with or without out-of-order execution, supporting a variety of pipelined execution mechanisms. With minor modifications, it can also handle out-of-order completion machines.

- The implementation machine is described at the micro-architectural level of abstraction, using uninterpreted functions to abstract away from uninteresting details. Also, no resource size constraints are assumed. Thus, our verification results are valid for any fixed size instantiation of the design.

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 demonstrated using PVS [40], a theorem prover from SRI International.\(^1\)

Our methodology is experimentally validated by verifying six examples with a wide variety of pipelined execution mechanisms:

\(^1\)We believe it can also be implemented in other verification systems that support uninterpreted functions and provide the necessary decision procedures.
• Two in-order execution and in-order completion machines are verified. The first example (Ex1.1) is a five-stage pipeline implementing a subset of the DLX architecture [23]. The second example (Ex1.2) is a dual issue version of the same architecture.

• Two out-of-order execution and in-order completion machines with a reorder buffer are verified. The first example Ex2.1 supports arithmetic instructions only. The second example Ex2.2 is substantially more involved with support for many micro-architectural features. It has three types of instructions namely arithmetic instructions, memory instructions and branch instructions, and some special instructions allowed only in a supervisory mode. It has multiple execution units, executing the instructions in an out-of-order manner. Branch instructions are executed speculatively with a very general branch predictor. Three types of exceptions are modeled namely arithmetic exceptions, "address out of bounds" exceptions for memory operations, and illegal instruction exceptions. All these exceptions are precise [23]. The design has a "store buffer" that records the values to be written into the data memory, and these are committed to the data memory at a later appropriate time.

• Two out-of-order completion machines are also verified. The first example Ex3.1 allows certain arithmetic instructions to bypass certain other arithmetic instructions when their destination registers are different. The second example Ex3.2 implements Tomasulo’s algorithm [54] without a reorder buffer, but modeling arithmetic instructions only.

The proofs in all these case studies are accomplished with reasonable manual effort. Each verification effort builds on the earlier efforts, reusing some of ideas and the proof machinery. Based on all the above observations, we argue that the completion functions approach is a viable and a promising approach to verify complex pipelined processor designs.

1.4 Related Work

Several researchers have attempted formal verification of non-pipelined as well as pipelined microprocessors of varying complexity. Notable examples of non-pipelined processor verification include Viper [15] by Cohn, FM8501 [32] by Hunt, and FM9001 [8] by Brock, Hunt and Kaufmann. One of the earliest studies in the context of pipelined
processor verification was carried out by Srivas and Bickford [50] in the MiniCayuga verification effort. Some of the other early verification efforts are described in [17,48,53,57].

One of the widely followed approaches in pipelined processor verification is the flushing approach, introduced by Burch and Dill [13]. They observed that the effect of flushing the pipeline can be used to compute a suitable abstraction function used in the correctness criterion. This can be achieved, automatically in many cases, by simulating the implementation machine without feeding a new instruction for a large enough number of cycles to finish the execution of all pending instructions. Burch and Dill used this flushing approach along with a validity checker [3,35] to effectively automate the verification of several processor designs, and the same approach has also been used in other verification efforts [9]. But this approach has the drawback of making the size of the abstraction function generated and the number of examined cases impractically large for complex superscalar pipelines, since it leads to a monolithic correctness formula. Another drawback is that this approach is hard to use for pipelines with indeterminate latency (which can arise, for example, when the control involves data dependent loops).

To verify a more complex design, Burch [12] introduced a “controlled flushing” technique, developed a decomposition of the correctness assertion into three subproblems, and used a validity checker that required fine-tuning with many manually assisted case splits. The soundness of the approach is proved separately by Windley and Burch [56]. It is unclear how these techniques scale up to verify a design with out-of-order execution. In [34], Jones, Skakkebæk and Dill introduced a two step “enhanced incremental flushing” technique to deal with out-of-order execution processors. While the entire proof was not mechanized, it has other drawbacks like making assumptions on the time taken by the execution units. It also does not handle speculative execution or exceptions.

Following a different line of research in [10], Bryant, German and Velev exploit some specific characteristics of the formula corresponding to the correctness criterion of the flushing approach while deciding its validity. They apply several conservative abstractions to the hardware model so that the correctness formula generated satisfies their constraints. Certain impressive verification results are reported [55], but their method is applicable to certain restricted class of pipelines only (ones using their “abstract memory model”) and has not been demonstrated for designs with out-of-order execution.

Sawada and Hunt have verified the correctness of an out-of-order execution processor model with a reorder buffer, a store buffer, speculative execution and exceptions [45,47].
They construct an explicit intermediate abstraction in the form of a table called MAETT (Micro-Architectural Execution Trace Table [46]) that represents the trace of all executed instructions up to the present time. They formulate several invariant properties over this intermediate abstraction and prove the final correctness from these invariant properties. Our approach avoids the construction of an explicit intermediate abstraction and takes significantly less effort to prove the correctness property. On an example comparable in complexity to EX2.2, they report taking 15 person months of effort [45], whereas our proof was accomplished in 34 person days.

Pnueli and Arons [43] have verified an out-of-order execution processor design with a reorder buffer and arithmetic instructions only by synchronizing the implementation and the specification machines at instruction retirement time. They have later extended their technique in [2] to handle speculation and exceptions (but with no data memory instructions). Their approach relies on formulating many invariant properties on the implementation machine but there is no systematic approach to generate these invariants and other obligations needed.

A verification of a processor model implementing Tomasulo's algorithm without a reorder buffer is carried out by Berezin, Biere, Clarke and Zu [5]. They use a technique combining symbolic model checking with uninterpreted functions. However, the verification is for a small fixed configuration of the processor model and proof decomposition is not emphasized. They later extend their work [6] to verify the model without any resource size bounds where they use a variant of the completion function idea we proposed. McMillan uses compositional model checking and aggressive symmetry reductions to manually decompose the proof of a similar processor model via refinement maps [37]. Setting up these refinement maps requires information similar to that provided by the completion functions. He has later extended his techniques to handle uninterpreted functions [38]. The processor model verified does not have speculative execution or exceptions. Yet another proof of a fixed size processor model implementing Tomasulo's algorithm without a reorder buffer is carried out by Henzinger, Qadeer and Rajamani using assume-guarantee reasoning [24].

Finally, Park and Dill have used idea of aggregation functions/completion functions in a different problem domain, that is, for distributed cache coherence protocol verification [41]. While the same notion of completion functions is used in our approach, the rest of the proof methodology is different with our work involving issues such as completing
the instructions in their program order, the squashing of instructions and the feedback logic, absent in their work/problem domain.

1.5 Organization of the Dissertation

Chapter 2 discusses the processor correctness criterion used in our work. Chapter 3 describes the completion functions approach as it applies to in-order completion machines. Chapters 4 and 5 describe the application of the methodology for in-order completion machines, without and with the out-of-order execution capability respectively. The modifications to handle out-of-order completion machines are explained in Chapter 6. Chapter 7 provides concluding remarks and discusses possible future work. Appendix A provides a brief introduction to PVS, Appendix B describes the modeling of the implementation and specification machines, Appendix C describes the PVS strategies used in the methodology, and finally Appendix D shows the PVS proof organization in all the examples described in this dissertation and an example PVS proof.
CHAPTER 2
MODELING PIPELINED PROCESSOR CORRECTNESS

This chapter discusses the microprocessor correctness criterion used in our work. It also discusses how the implementation and the specification machine descriptions are provided, and how the different proof obligations are formulated in our approach. Since the methodology is implemented using PVS, the syntax and notations of the PVS language are used throughout the dissertation. Refer to Appendix A for a brief description of PVS.

2.1 Machine Descriptions

A pipelined implementation $I$ of a microprocessor (at the micro-architectural level of abstraction) is given in the form of its transition function $^1$ denoted by $I\_step$. The implementation machine state is made up of two types of components, namely the observable components and the nonobservable components. The observable components are those visible to a programmer such as the register file and the data memory, while the nonobservable components are used to store the intermediate values in the pipelined execution of the instructions. $I\_step$ states how the next-state of the implementation machine is obtained in terms of the present state and the present input values, by defining a function $next\_ic$ for each implementation component $ic$ that describes how its new value is obtained. All implementation machines considered in this dissertation take one input signal $no\_op$?. In a cycle in which $no\_op$? is asserted, no new instructions are inserted/issued into the pipeline. The instructions in the pipeline eventually get executed, compute their results, commit them to the observable components and get retired. The instructions fetched speculatively and found to be mispredicted, and those following an instruction that has raised an exception are squashed without affecting the observable

$^1$The implementation machine is deterministic. However the signal definitions in the implementation machine are allowed to be underspecified.
components (except for correcting the program counter). The instructions executing in the pipeline but not yet retired are referred to as partially executed instructions or unfinished instructions. The state where the pipelined machine has no partially executed instructions is called a flushed state, and the predicate Flushed? characterizes such states. And the initial states of the implementation machine are characterized by the predicate Init. The signatures of I_step, Flushed? and Init along with the types of their arguments are shown in 1. Refer to Appendix B for some more details about the implementation machine modeling.

\[
\text{state}_I: \text{TYPE} = [\# \text{observable}_1 : \text{obs}_1, \\
... \\
\text{observable}_k : \text{obs}_k, \\
\text{nonobservable}_1 : \text{nonobs}_1, \\
... \\
\text{nonobservable}_n : \text{nonobs}_n \#] \\
\text{inputs}_\text{type}: \text{TYPE} = [\# \text{no}_\text{op}? : \text{bool} \#] \\
\text{I}_\text{step} : [\text{state}_I, \text{inputs}_\text{type} \to \text{state}_I] \\
\text{Flushed?} : [\text{state}_I \to \text{bool}] \\
\text{Init} : [\text{state}_I \to \text{bool}]
\]

A specification \(\mathcal{S}\) of the microprocessor is also given which describes the programmer visible semantics, that is, the Instruction Set Architecture (ISA) semantics of the implementation machine. The specification machine state is made up of the observable components of the implementation machine. The specification is provided in the form of a transition function \(A_{\text{step}}\) describing the effect on the observable components of executing a given instruction from the instruction set in one clock cycle, and is specified in a form similar to that of \(I_{\text{step}}\). The function projection extracts the values of the observable components given an implementation machine state. The signatures of \(A_{\text{step}}\) and \(\text{projection}\) are shown in 2. Again, Appendix B has some more details on the specification machine modeling.

\[
\text{state}_A : \text{TYPE} = [\# \text{observable}_1 : \text{obs}_1, \\
... \\
\text{observable}_k : \text{obs}_k \#] \\
A_{\text{step}} : [\text{state}_A \to \text{state}_A] \\
\text{projection} : [\text{state}_I \to \text{state}_A]
\]
2.2 Correctness Criterion

Intuitively, a pipelined implementation of a processor is correct if the behavior of the implementation machine starting in a flushed state, executing a program and terminating in a flushed state is emulated by the ISA level specification machine whose starting and terminating states are in direct correspondence through projection. This criterion is shown in Figure 2.1(a) where \( n \) is the number of implementation machine transitions in a given run of the machine and \( m \) corresponds to the number of instructions executed as per the specification machine by this run. (It is not necessary to explicitly relate the instructions in the two sequences shown in the figure, since the instruction memory and the program counter are part of the implementation and the specification machines in our work.)

The criterion shown in Figure 2.1(a) can be proved by an easy induction on \( n \) once the criterion shown in Figure 2.1(b), called the commutativity obligation, is proved on a single implementation machine transition and a certain projection obligation discussed below is established. The commutativity obligation states that if the implementation machine starts in an arbitrary state \( q \) and the specification machine is in a corresponding specification state given by an abstraction function \( \text{ABS} \), then after executing their respective transitions the new states obtained correspond (according to the same abstraction function \( \text{ABS} \)).\( \text{ABS} \) must be chosen so that for all flushed states \( fs \) the projection obligation \( \text{ABS}(fs) = \text{projection}(fs) \) holds. The commutativity obligation uses a modified specification transition function \( \text{A\_step\_new} \), which denotes zero or more applications of \( \text{A\_step} \) (the number of applications of \( \text{A\_step} \) is specified by an argument to \( \text{A\_step\_new} \)), because an implementation transition from an arbitrary state might correspond to executing in the specification machine zero or more instructions. The number of instructions executed

![Diagram](image-url)

Figure 2.1. Illustration of the pipelined microprocessor correctness criterion
by the specification machine corresponding to an implementation transition is provided by a user-defined synchronization function \texttt{sync\_fun}, and this number depends on how the abstraction function is defined and the number of instructions issued/retired in the given implementation transition.

In proving the commutativity and the projection obligations, it is sufficient to consider only those implementation states actually reachable in some execution of the implementation machine. Indeed, these obligations may not hold for some states not actually reachable in any execution of the implementation machine. To rule out considering such states, the user needs to discover invariant properties to restrict the set of implementation states considered in the proof of these obligations. The predicate \texttt{Reachable?} is defined to be the conjunction of these invariant properties, and it characterizes a superset of the legal reachable states. It is necessary to prove that this set contains the initial states of the implementation machine and that it is closed under \texttt{I\_step}. \ref{table} shows the functions, predicates and obligations introduced above.

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{ABS:} [ state\_I $\rightarrow$ state\_A] \\
\textbf{sync\_fun:} [ state\_I, inputs\_type $\rightarrow$ nat] \\
\textbf{A\_step\_new:} [ state\_A, nat $\rightarrow$ state\_A ] \\
\textbf{Reachable?:} [state\_I $\rightarrow$ bool] \\
\textbf{Reach\_state\_I:} TYPE = \{ q : state\_I $|$ Reachable?(q) \} \\
\hline
\end{tabular}
\caption{Implementation States and Predicates}
\end{table}

\textbf{commutativity: THEOREM}
\begin{align*}
\forall (q:\text{Reach\_state\_I},i:\text{inputs\_type}) : \\
\text{ABS}(\text{I\_step}(q,i)) = \text{A\_step\_new}(\text{ABS}(q),\text{sync\_fun}(q,i))
\end{align*}

\textbf{projection: THEOREM}
\begin{align*}
\forall (q:\text{Reach\_state\_I}) : \\
\text{Flushed?}(q) \implies \text{ABS}(q) = \text{projection}(q)
\end{align*}

\textbf{Inv\_Init: THEOREM}
\begin{align*}
\forall (q:\text{state\_I}) : \\
\text{Init}(q) \implies \text{Reachable?}(q)
\end{align*}

\textbf{Inv\_Closure: THEOREM}
\begin{align*}
\forall (q:\text{Reach\_state\_I},i:\text{inputs\_type}) : \\
\text{Reachable?}(\text{I\_step}(q,i))
\end{align*}

It is now proved that the commutativity obligation and the projection obligation together imply the correctness criterion expressed in Figure 2.1(a). Say a sequence of \texttt{n} implementation transitions starting at state \texttt{I}_0 and ending in state \texttt{I}_n (going through \texttt{I}_1 \ldots \texttt{I}_{n-1}) is given. It can be shown by a straight forward induction on \texttt{n} and using the
commutativity obligation that, there is a sequence of m specification transitions starting at state $S_0$ and ending in $S_m$ where $S_0$ is $\text{ABS}(I_0)$, $S_m$ is $\text{ABS}(I_n)$ and m is the sum of the numbers returned by the synchronization function at each of states $I_0 \ldots I_{n-1}$. Now if $I_0$ and $I_n$ were flushed states, then from the projection obligation, $S_0$ is $\text{projection}(I_0)$, $S_m$ is $\text{projection}(I_n)$ and the correctness criterion of Figure 2.1(a) follows.

Observe that the abstraction function is used only in the intermediate proof obligation, that is, the commutativity obligation. The final correctness criterion shown in Figure 2.1(a) does not use the abstraction function at all. So, as long as the commutativity obligation and the projection obligation are proved, the correctness criterion shown in Figure 2.1(a) follows, whatever be the definition of the abstraction function. The questions such as “Is the abstraction function correct?” “What does the commutativity obligation mean if the abstraction function is incorrect?” and “Can a buggy abstraction function fix a bug in the implementation?” are all irrelevant.

Mechanical formal verification of the obligations formulated in [3] is the focus of this dissertation. The most difficult problem here is to define a suitable abstraction function mapping an implementation state to a specification state, and to discover the necessary invariant properties. The completion functions approach primarily addresses the problem of defining an abstraction function, while also providing some guidelines for the discovery of invariant properties. The definition of the abstraction function depends on how the implementation and the specification machines are synchronized. Two very natural choices for doing this are to synchronize them at instruction issue time (when new instructions are being inserted into the pipeline) or to synchronize them at instruction retirement time (when some instructions are being retired from the pipeline). When the implementation and specification machines are synchronized at instruction issue time, a suitable abstraction function can be defined by flushing the pipeline, that is, by completing the execution of all the pending instructions in the pipeline ( [50], [13]).

The completion functions approach synchronizes the two machines at the instruction issue time and suggests a way of defining an abstraction function in a manner that leads to an elegant decomposition of the proof. The alternative approach of synchronizing the machines at instruction retirement time is used in other works (e.g., [43]). The study of the tradeoffs between these two methods of synchronizing is not done in this dissertation.
CHAPTER 3

THE COMPLETION FUNCTIONS

APPROACH

The key step in proving the correctness of pipelined microprocessors is to discover a formal correspondence between the execution of the implementation and the specification machines. As described in Chapter 2, this correspondence can be achieved by defining a suitable abstraction function from implementation states to specification states. The completion functions approach develops a systematic way of constructing this abstraction function in a manner that leads to an efficient decomposition of the proof. When the implementation and specification machines are synchronized at instruction issue time, a suitable abstraction function can be defined by flushing the pipeline, that is, by completing the execution of all the pending instructions in the pipeline. Our approach synchronizes the two machines at instruction issue time. Hence the abstraction function in our approach maps a given implementation state to a state where all the unfinished instructions in the pipeline are completed and then extracts the values of the observable components. Section 3.1 describes the different steps in constructing such an abstraction function. Section 3.2 discusses how to decompose the proof of correctness, the process of generating the different verification conditions, and techniques for discharging the verification conditions.

The completion functions approach is primarily applicable to in-order completion machines, that is, those machines where the executing instructions commit their results to any given observable component other than the program counter\(^1\) in their program order. The approach can be extended to handle out-of-order completion machines too, that is, those machines where the executing instructions may possibly commit their results

\(^1\)Even in in-order completion machines, the program counter may be updated out-of-order by the instructions, albeit speculatively. This is due to its role as a counter, keeping track of the next instruction to be fetched, and due to its being the target of branch instructions. This point is revisited later in the chapter.
to some or all the observable components out of their program order. This chapter concentrates on in-order completion machines only, and the extensions necessary to handle out-of-order completion machines are described in Chapter 6.

3.1 Compositional Construction of the Abstraction Function

As pointed out earlier, the abstraction function in the completion functions approach expresses the cumulative effect of flushing the pipeline. Such an abstraction function can be constructed by following the three steps listed below, each of which is outlined in the subsequent sections:

- Determine the program order of the unfinished instructions.
- Define a completion function for every unfinished instruction.
- Construct the abstraction function using these completion functions.

The final section discusses how to define the synchronization function in our approach.

3.1.1 Finding the unfinished instructions and their program order

A pipelined implementation, at any given time, has a set of partially executed instructions and a correct implementation is supposed to complete them in their program order. (The program order is defined by the ISA level specification machine.) The first step in constructing the abstraction function is to determine these partially executed instructions in the pipeline and their program order. Most in-order completion machines can be classified into two categories based on whether or not they allow the instructions to be executed out-of-order: in-order execution machines and out-of-order execution machines. These machines (when working correctly) maintain the executing instructions in the pipeline in their program order\(^2\) implicitly or explicitly. Accordingly, the following two classes of pipelines are considered which include many of the commonly found micro-architectural designs [33]:

\(^2\)When speculative execution is entertained, the order in which the instructions are maintained corresponds to the program order only if none of the executing instructions are mispredicted. For ease of explanation, this detail is ignored and the instructions are referred to as being just stored in their program order.
• **Class 1** pipelines (supporting in-order execution and in-order completion): These pipelines have several stages through which an instruction flows and there is a structural ordering on these pipeline stages. There is a set of pipeline latches separating two successive pipeline stages, each of which may hold a partially executed instruction. These pipelines hold a small and fixed number of instructions. The structural ordering of the pipeline stages implicitly defines the program order of the executing instructions, that is, instructions occurring towards the “end of the pipeline” are earlier in the program order. An example is the DLX pipeline of Hennessey and Patterson [23].

• **Class 2** pipelines (supporting out-of-order execution and in-order completion): The implementation machine stores the unfinished instructions in their program order explicitly by using a reorder buffer [33]. Every instruction executing in the processor has a reorder buffer entry allocated to it, and the pipeline can hold as many instructions as the size of the reorder buffer.

In both these classes of pipelines, the program counter is an observable component that may be modified out-of-order by the instructions, albeit speculatively. The program counter keeps track of the next instruction to be fetched, and hence is incremented or updated according to a speculated branch target whenever a new instruction is issued. An already issued instruction (which is earlier in the program order than the new instructions issued) modifies (and corrects) the program counter later if it turns out to be a mispredicted branch or if it raises an exception. In such a case, all the subsequent instructions in the pipeline are squashed. Also, the already issued instructions modify the program counter in program order only.

### 3.1.2 Specifying the completion functions

Having determined the program order of the unfinished instructions (or what ought to be the program order in a correctly working implementation), the second step is to define a completion function for every unfinished instruction in the pipeline. Each completion function specifies the *desired effect* on the observable components of completing a particular unfinished instruction assuming those that are ahead of it (in the program order) are completed. That is, it specifies what ought to be the effect on the observable components of completing a given unfinished instruction in an atomic fashion. The
completion functions, which map an implementation state to an implementation state, leave all nonobservable state components unchanged. Our methodology requires the user to provide the definitions of these completion functions and a general method for doing so is described below. However, in some cases, these definitions can be derived from the implementation machine using symbolic simulation. While this technique helps reduce the manual effort needed in specifying the completion functions, it has the disadvantage of potentially delaying the discovery of design bugs (details elaborated later in Section 4.1.6).

Not every instruction in the pipeline gets executed completely and updates the observable components. If an instruction raises an exception or if the target address is mispredicted for a branch instruction, then the instructions following it are squashed. To specify this behavior, a squashing predicate is defined for every unfinished instruction. The squashing predicate captures the conditions under which a given unfinished instruction can cause the subsequent instructions (in the program order) to be squashed. The completion function for a given instruction will update the observable components only if it is not squashed by any of the instructions preceding it.

We now elaborate on specifying the completion functions and the squashing predicates. An instruction executing in the pipeline passes through many different phases such as issued, dispatched, executed, etc. before being retired or squashed. A given unfinished instruction is in one of these phases at any given time and the information about this instruction (the source values, destination register, etc.) is held in the various implementation components. All the different possible phases for an unfinished instruction are identified. For each instruction phase “ph”, the following are defined: a predicate “Instr\_ph?” that describes whether a given instruction is in phase “ph”, a function “Action\_ph” that specifies the desired effect of completing an instruction in that phase, and a predicate “Squash\_rest\_ph?” that gives the conditions under which an instruction in that phase can squash all the subsequent instructions (in the program order).

Consider Class 1 pipelines. Each set of pipeline latches holds an unfinished instruction in a certain unique phase of its execution, and hence these sets of latches can be used to identify the different phases. Since the unfinished instructions reside in these sets of latches, they can also be used to identify the different instructions. So, an unfinished instruction as well as the phase it is in is identified by the set of pipeline latches it resides in. (This makes the definition of “Instr\_ph?” predicates trivial.) The template for a completion function is shown in \[1\]. stage\_i stage\_j stands for the set of pipeline
latches between the two pipeline stages \texttt{stage}_i and \texttt{stage}_j. When \texttt{kill?} argument is \texttt{TRUE}, the completion function will not update any of the observable components. Since an instruction is identified by the set of pipeline latches it resides in, which also identifies a unique phase, the squashing predicates for the different instructions will just be the corresponding \texttt{``Squash_rest?_phi''} predicates.

```java
Complete_stage_i_stage_j(q:state_I, kill?:bool): state_I =
    IF kill? THEN q
    ELSE Action_stage_i_stage_j(q) END IF
```

Now consider \texttt{Class 2} pipelines. An unfinished instruction is identified with the index \texttt{rbi} of the reorder buffer entry allocated to it. A single parameterized completion function and a single parameterized squashing predicate applicable to all the unfinished instructions in the reorder buffer is defined, the templates of which are shown in \texttt{2}.

```java
Complete_instr(q:state_I, rbi:rbindex, kill?:bool): state_I =
    IF kill? THEN q
    ELSEIF Instr_phi?(q,rbi) THEN Action_phi(q,rbi)
    ELSEIF ... Similarly for other phases ...
    ELSE q END IF

Squash_rest?_instr(q:state_I, rbi:rbindex): bool =
    IF Instr_phi?(q,rbi) THEN Squash_rest?_phi(q,rbi)
    ELSEIF ... Similarly for other phases ...
    ELSE FALSE END IF
```

Observe that the templates for \texttt{Class 1} pipelines is a special case of the templates for \texttt{Class 2} pipelines, though they appear different. This is because an unfinished instruction as well as the phase it is in is identified by the set of pipeline latches it resides in for \texttt{Class 1} pipelines. Since \texttt{Class 1} pipelines have a small and fixed number of unfinished instructions, a completion function is defined per unfinished instruction as opposed to the other alternative of defining a single parameterized completion function that applies to all the unfinished instructions. This second alternative is used for \texttt{Class 2} pipelines because the number of unfinished instructions is determined by the size of the reorder buffer in these pipelines (an uninterpreted constant in our models).

The completion function and the squashing predicate for a given instruction are specified under the assumption that the instructions ahead of it are all completed. That is, it is assumed that the argument \texttt{q} to these functions for a given instruction corresponds to the state where all the instructions ahead of that given instruction are completed. This
assumption is met by composing these completion functions in their program order while constructing the abstraction function (described later in Section 3.1.3).

Specifying the completion function and the squashing predicate for a given instruction requires one to: (i) understand how the information about that instruction is stored in the implementation machine, and (ii) understand how they are to be combined to have the effect of completing that instruction. We believe that a typical designer has a very clear understanding of these two aspects and will not find the task of specifying the completion functions and the squashing predicates onerous. Any mistakes, either in specifying the completion functions/squashing predicates or in constructing the abstraction function, may lead to a false negative verification result, but never a false positive (as long as the commutativity obligation and the projection obligation are proved).

In the specification of the “Action^ph” functions and “Squash^rest^ph” predicates, the expected values of the source operands of an instruction need to be specified, when the instruction is in a phase where its source operands are not yet ready. Our solution is to simply read them from the observable components. The justification for this being correct is based on the assumption made while defining the completion functions/squashing predicates mentioned earlier. That is, a given instruction is completed in a context where the instructions ahead of it are all completed, and hence all its source operands ought to be available in the observable components.

3.1.3 Constructing the abstraction function

The final step is to construct the abstraction function by composing these completion functions in their program order, and then extracting the values of the observable components. In constructing the abstraction function in the above manner, the kill? argument of the completion functions for all the unfinished instructions is to be specified (Refer to the templates in [1] and [2]). The kill? argument for the completion function of a given instruction is TRUE when the squashing predicate is TRUE for any of the instructions ahead (in the program order) of that given instruction. This definition of the abstraction function leads to a very natural decomposition of the proof of the commutativity obligation and supports incremental verification.

3.1.4 Defining the synchronization function

In the completion functions approach, the implementation and the specification machines are synchronized at instruction issue time (i.e., the time when an instruction
enters the pipeline). The abstraction function maps a given implementation machine state to a specification machine state by completing all the executing instructions in the pipeline. The synchronization function specifies the number of instructions executed by the specification machine corresponding to a given implementation machine transition, so as to keep the specification machine synchronized with the implementation machine after its transition. The number returned by the synchronization function depends on factors such as whether or not there is room in the processor to fetch a new instruction, whether or not no_op? is asserted, whether or not the squashing predicate is TRUE for any of the executing instructions in the pipeline, and whether or not multiple instructions can be issued into the pipeline in a single implementation transition.

3.2 Decomposing the Proof of the Commutativity Obligation

This section illustrates the key idea in decomposing the proof by means of an example and describes a general method for doing so later. The correctness issues in verifying a pipelined processor can be divided into the following three broad categories: correctness of the forward movement of an instruction as it flows down the pipeline, correctness of the squashing of instructions in case of exceptions/mispredictions etc, and the correctness of the source operands of the instructions including the correct backward movement of data via the feedback logic in case of dependencies. The way our approach addresses these correctness issues is pinpointed.

3.2.1 Illustration on a simple example

Consider a pipeline with k unfinished instructions 1 ... k in program order. Instruction 1 is earliest in the program order and k is the latest. All instructions update the observable regfile when they are retired. Let the completion functions of these unfinished instructions be c_1 ... c_k, respectively. Assume I_step retires instruction 1 and fetches a new instruction k+1. So the pipeline has instructions 2 ... k+1 in that order in state I_step(q,i). Assume that none of the executing instructions raise any exceptions.

The commutativity obligation in this case is depicted in Figure 3.1. i1 is an arbitrary reachable implementation state and i2 is the state obtained after an implementation transition. The abstraction function is defined to complete all the pending instructions in the pipeline, and then to extract the values of the observable components. So in state i1, the abstraction function completes the instructions 1 ... k in that order, and in state
Figure 3.1. Proof decomposition illustrated on a simple example

i2, it completes the instructions 2 \ldots k+1 in that order. s1 is the specification machine state that corresponds to i1 (correspondence defined using the abstraction function), and s2 is the state obtained after a specification machine transition from s1. The goal is to show that i2 and s2 are a pair of corresponding states. All this is depicted in Figure 3.1: the rectangles and the small circles represent implementation machine states and the large circles on the right hand side represent specification machine states. (projection is not explicitly shown in the figure.) The convention of using arrows (one sided) is interpreted as follows: the arrow from p1 to p2 labeled C_2 implies that state p2 is obtained when the completion function C_2 is applied in state p1. In Figure 3.1, the states occurring on the path “i1 \rightarrow i2 \rightarrow s2” are said to be occurring on the implementation side of the commutativity obligation (since I_step occurs on this path), and the states occurring on the path “i1 \rightarrow s1 \rightarrow s2” are said to be occurring on the specification side of the commutativity obligation (since A_step occurs on this path).

One way of proving the commutativity obligation is by showing that the states obtained after traversing the two paths “i1 \rightarrow s2”—via s1 and via i2—are the same. Due to our construction of the abstraction function as a composition of completion functions, this obligation can be decomposed into a series of verification conditions as shown in Figure 3.1. Each verification condition VC_i asserts that the contents of regfile is the same in the two states they point to in Figure 3.1. Since I_step retires instruction 1 and updates regfile, it is expected that the contents of regfile at state i2 is the same as after completing instruction 1, that is, at state p1, and this is captured by VC1. Similarly regfile contents should be same at states p2 and p3 as captured by VC2 because both these correspond to states where instructions 1 and 2 are completed in that
order. Similarly other verification conditions can be justified.

Now consider proving these verification conditions. We consider the “instruction phase transitions,” that is, how an instruction makes a transition from its present phase to its next phase when the implementation machine makes a transition, and use this in generating the different cases for proving a verification condition. Consider proving VC3, which is relating the effect of completing instruction 3 in states p2 and p3. Imagine that instruction 3 is in issued phase in state i1 and it goes to dispatched phase after an implementation transition (i.e., in i2). Completing any instruction only affects the observable components (in particular, it has no effect on the phases of the instructions that occur later in the program order), and hence instruction 3 is in issued phase in p2 and in dispatched phase in p3. So to complete instruction 3, Action\_issued can be used in p2 and Action\_dispatched can be used in p3. It is then shown that these have the same effect on regfile. Proving this case of VC3 thus verifies a part of the implementation machine that moves an instruction from issued phase to dispatched phase. Similarly, the remaining cases prove other aspects of the instruction flow in the pipeline. In fact, the proof builds up in a layered manner verifying that the implementation correctly completes an instruction in the “last” phase against its specification (the corresponding “Action” function), and then using this “correct” specification in verifying that the implementation correctly completes an instruction in the penultimate phase, and so on.

The proof of VC3 discussed above uses VC2 because when the “Action” functions are expanded, the terms regfile(p2) and regfile(p3) appear in the formula corresponding to VC3 (on the specification and the implementation sides respectively; these notions are similar to that of the commutativity obligation), and these two are equal by VC2. Since only one instruction is considered at a time and because the verification conditions about the previous instructions are used, the sizes of the expressions generated during the proof and the required case-analysis while checking the validity of the formula corresponding to the verification condition are kept under control. The verification proceeds incrementally, debugging the different aspects of the pipeline separately. This makes it easier to locate errors, thus facilitating debugging. 3

---

3In some of the approaches that generate a single monolithic verification condition( [13], [10]), if there is a design bug in the pipeline, the verification method (if at all applicable) would produce a counterexample—a set of formulas potentially involving all the implementation components—that implies the negation of the formula corresponding to the verification condition. Such a counterexample cannot easily pinpoint the stage in which the design bug occurs.
3.2.2 Generating and proving the verification conditions

Section 3.2.2 and Section 3.2.3 discuss the general guidelines to be used in generating and proving the verification conditions in our approach, applicable to the two classes of pipelined processors discussed earlier. It is not claimed that these guidelines are sufficient; however, they are adequate for a variety of pipelined processor designs, including all those designs considered in this dissertation. The verification conditions formulate certain properties about the implementation machine that ought to hold in a correct implementation. The key idea in proving the commutativity obligation is to show that all the instructions in the pipeline complete their execution correctly. So verification conditions are formulated that relate the effect of instructions completing one by one in the present and the next implementation states until all the instructions are considered. The final goal in our approach is to prove the commutativity obligation (and the projection obligation). The verification conditions discussed below act as “helpers” in managing the expression size and number of cases examined, and lead up to the proof of the commutativity obligation. However, it is not mandatory to formulate and prove every single verification condition discussed below. The soundness of our approach is not compromised as long as the commutativity obligation is proved, whether or not, one formulates and proves all the verification conditions that are discussed below. In practice however, without the verification conditions, the proof of the commutativity obligation is very hard.

Figure 3.2 depicts the instructions in a processor and their phases, before and after an implementation transition (one possible scenario). X represents the instruction(s) that retire and update the observable components in the current transition, N represents the instruction(s) newly fetched in the current transition, and the small circles represent the instructions that are still being executed (but may have changed phase during the current transition). The instructions are numbered for reference in the discussion below.

The first kind of verification conditions relate the effect on the observable components of completing instructions till a given instruction (that is, from instruction 1 till the given instruction in program order) in the present and the next implementation states. The second kind of verification conditions relate the truth values of the squashing predicate for a given instruction in the present and the next implementation states. Coming back to the first kind of verification conditions, in an in-order completion machine, though the instructions may be executed out-of-order, they are committed to the observable com-
Figure 3.2. Depicting the instructions and their phases, before and after an implementation transition

ponents (except the program counter) in their program order. This observation suggests the following verification conditions, one for every instruction in the pipeline: that the effect on an observable component (other than program counter) of completing all the instructions in the program order till a given instruction $i$ (that is, from instruction 1 till the given instruction $i$) ought to be the same in the present and the next implementation states. The second kind of verification conditions assert that the squashing predicate for a given instruction has the same truth value in the present and the next implementation states. These verification conditions are slightly modified under certain circumstances listed below:

1. Consider the case when an instruction (say $i$) squashes all the subsequent instructions in the current implementation transition. Then the squashing predicate ought to be TRUE in the present implementation state for instruction $i$. For the instructions that come after the instruction $i$, the two verification conditions discussed above are formulated with a precondition: that the instruction $i$ is not squashing the subsequent instructions in the current implementation transition. (The subsequent instructions are not present in the pipeline in the next implementation state, when squashed. 4)

2. If an instruction other than the earliest one in the program order can update (i.e.,

---

4 In certain implementations, such instructions may remain in the pipeline in an invalid phase. The completion functions in such a case will not modify any of the observable components, and the modification described for the verification conditions may not be necessary.
commit to) a certain observable component in an implementation transition, then the first verification condition above holds for that observable component when the number of instructions considered in the verification condition is large enough to include the instruction which modifies that observable component. For the earlier instructions, the first kind of verification conditions assert that the observable component remains unchanged. In general, the first kind of verification conditions may be different for different observable components.

As a concrete illustration, assume that instruction 3 in Figure 3.2 modifies an observable component, say rf, in the current implementation transition, but instructions 1 and 2 do not. Then the first kind of verification conditions hold for rf for instructions numbered 3 and higher only. Completing the instructions till 1 and completing the instructions till 2 ought not modify rf. If either instruction 1 or instruction 2 could modify rf, that violates the in-order completion semantics.

3. If multiple instructions can update a certain observable component in a single implementation transition, then the first kind verification conditions holds for that observable component only when the number of instructions considered includes all of the instructions updating the observable component.

As a concrete illustration, assume that instructions 1, 2 and 3 in Figure 3.2 modify an observable component, say rf, in the current implementation transition. Then the first kind of verification conditions hold for rf for instructions numbered 3 and higher only.

4. For any read-only observable component (instruction memory in our models), a simpler verification condition is generated (of the first kind) that states that the executing instructions ought not modify such an observable component at all.

So far, generating the verification conditions for the program counter is not discussed. For the program counter, it is not possible to relate its value in the present and the next implementation states by considering the effect of completing the instructions one at a time in their program order. This is because a new instruction entering the pipeline during an implementation transition modifies the program counter either by incrementing it or by updating it according to the speculated target address, and in both cases, this new instruction is the latest one in the program order.
However, verification conditions similar to those of the first kind for any other observable can be generated for the program counter under certain conditions. The program counter values can be related in the present and the next implementation states when squashing predicate is TRUE for one of the instructions considered in the verification condition (i.e., one among the instruction 1 till the given instruction). Such an instruction, when completed, updates the program counter and the subsequent instructions are not of interest, and it ought to update the program counter in the same way in both the present and the next implementation states. Even when the squashing predicate is not TRUE for any of the instructions considered, the above verification condition holds if there is no room in the pipeline to fetch a new instruction or if \texttt{no_op?} is asserted. Also, an instruction executing in the pipeline, when completed, modifies the program counter only if it causes the rest of the instructions to be squashed, that is, when the squashing predicate is TRUE for the instruction. This observation leads to the verification condition that completing the instructions till a given one does not modify the program counter if the squashing predicate is FALSE for all of them.

The verification conditions so far are pertaining to the instructions already executing in the processor. The verification conditions for the new instruction issued into the pipeline in the current transition are generated in a similar manner except that they are “matched” by a specification machine instruction (with the specification machine transition function acting as the completion function) in the present implementation state.

An optimization that can be used in formulating these verification conditions is the following: If the squashing predicate is TRUE for an instruction \texttt{i} in the present state, then it is TRUE in the next state too, and hence all the subsequent instructions will be killed in both the present and the next states. So in generating the verification conditions for the instructions that come after \texttt{i}, the squashing predicate is assumed to be FALSE of \texttt{i}. Another variation is to consider an exclusive and exhaustive set of cases (for example, \texttt{C} and \texttt{NOT C}, where \texttt{C} is some boolean condition) and then follow the guidelines described above in each of these cases.

Some specific details regarding the generation of the verification conditions for the two classes of pipelines considered in this dissertation are elaborated. Since the number of instructions is small and fixed in Class 1 pipelines, the verification conditions and the different cases due to the “instruction phase transitions” are generated individually. Each verification condition considers the effect on the observable components and the
truth value of the squashing predicate when an additional instruction is completed over and above those completed by the previous verification condition. Since Class 2 pipelines have a large number of instructions, a single parameterized verification condition applicable to all the pending instructions in the reorder buffer is generated. It is then proved by a simple induction argument where the induction hypothesis captures the effect of completing all the earlier instructions. The inductive argument is split into different cases based on how an instruction changes phase with an implementation transition.

The proofs of these verification conditions proceed using a standard strategy. They begin by considering the effect of only the outermost instruction on both sides of a verification condition. The resulting expression is simplified using techniques based on decision procedures and rewriting, and the verification conditions about the earlier instructions are used in the simplification process. The proofs build up in a layered manner as illustrated in an earlier section.

Proving these verification conditions establish the correctness of the first two issues mentioned in the beginning of Section 3.2, namely, the correctness of the forward movement of an instruction as it flows down the pipeline and the correctness of the squashing of the instructions in case of exceptions/mispredictions etc. The next section considers the third issue of the correctness of the source operands of the instructions.

### 3.2.3 Correctness of the source operands of the instructions

Recall that we had to specify how an instruction obtains its source operands when it is in a phase where the operands are not yet ready. As discussed earlier, our solution is to specify them as the values obtained when the corresponding observable components are read after all the instructions ahead of the given instruction are completed. Now the source operands as given by these specifications ought to remain the same whatever phase the instruction is in, and this observation gives rise to the third kind of verification conditions in our approach. That is, if an instruction changes phase from “ph1” to “ph2” in an implementation transition, the source operands for it as given by the specification for phase “ph1” in the current implementation state ought to be the same as the source operands for it as given by the specification for phase “ph2” in the next implementation state.

A closely related verification condition is for the case when the implementation actually obtains the source operands in the current implementation transition. This can happen
either when the instruction has no dependencies and it obtains the operand from the corresponding observable component or in case of dependencies, the “producer” instruction finishes its execution and forwards its result to the waiting instructions via the feedback logic. In either case, the source operand obtained in the current implementation transition ought to be equal to the expected value as given by the specification for the phase the instruction was in before the transition. Observe that this verification condition acts as a succinct correctness criterion for the feedback logic. The proofs of these verification conditions require one to argue that the implementation machine maintains the data dependencies among the executing instructions correctly.

3.2.4 Proving the commutativity obligation

The key idea in proving the commutativity obligation is to show that all the instructions in the pipeline complete their execution correctly. The three kinds of verification conditions about an unfinished instruction discussed earlier are used in proving the verification conditions about the next instruction, which are in turn used in proving the verification conditions about the subsequent instructions, eventually leading up to the proof of the commutativity obligation. The synchronization function specifies the different cases and the number of instructions executed by the specification machine in those cases, corresponding to an implementation transition. The proof of the commutativity obligation is split into different cases based on these cases of the synchronization function, and also on a per observable basis.

3.2.5 Invariant properties

Our approach may need certain invariant properties to be discovered by the user to constrain the set of states considered in the proof of the commutativity obligation. Typically these invariant properties are some example specific properties that ought to hold in a correct implementation, and are necessary to accomplish the proof of the commutativity obligation. The need for some of these invariant properties is obvious in our approach. For example, for Class 2 pipelines, having identified the different possible phases for an instruction, it must be shown that one and only case applies at any time in the definitions of Complete_instr and Squash_rest?_instr shown in [2], and this is established as an invariant property. However, the discovery of many other invariant properties is a manual process, and there is no global strategy for discovering them.
CHAPTER 4

PROCESSORS WITH IN-ORDER
EXECUTION AND IN-ORDER
COMPLETION

This chapter explains how to apply our methodology to verify two Class 1 examples, that is, with in-order execution and in-order completion. The first example Ex1.1 is a pipelined implementation of a subset of the DLX processor [23] and the second one Ex1.2 is a dual-issue version of the same example. We describe how to specify the completion functions and to construct the abstraction function, and illustrate the decomposition of the proof into verification conditions, and the proof strategies used in discharging these verification conditions. A technique that is aimed at reducing the manual effort in specifying the completion functions is also explained and illustrated on Ex1.1.

4.1 Application to Ex1.1

4.1.1 Details of Ex1.1

The implementation uses a five stage pipeline as shown in Figure 4.1. The five stages are Instruction Fetch IF, Instruction Decode ID, Execute EX, Memory Operations MEM and Writeback Results WB, and the intended functionality of each of the stages is also shown in the figure. The implementation machine state is made up of 19 implementation components, 15 nonobservable components and four observable components. The 15 nonobservable components are organized as four sets of pipeline latches, each set separating two consecutive pipeline stages and holding information about a partially executed instruction. The set of pipeline latches separating IF and ID stages will be referred to as "IF/ID latches" and other sets of pipeline latches are named similarly. The four observable components are the register file rf, the data memory dmem, the program counter pc and the instruction memory imem. The observable rf is updated in WB stage, dmem is updated in MEM stage, pc is updated in IF and ID stages, and imem is not modified at all. The Arithmetic Logic Unit in the EX stage is modeled as an uninterpreted function alu. The
Figure 4.1. Pipelined implementation of EX1.1
register file and the data memory are modeled as explained in Appendix B with select and assign operations for rf, and read and write operations for dmem respectively. The implementation supports six types of instructions: two arithmetic instructions alu_reg and alu_immed, two memory instructions load and store, and two branch instructions jmp and beqz, and their semantics are as explained in Appendix B.

In the absence of any dependencies, the executing instructions flow down the pipeline with every clock cycle passing through the successive pipeline stages. The implementation provides forwarding of data to the ID stage where the source operands of the instructions are read. If the instruction in the IF/ID latches is dependent on a load instruction in the ID/EX latches, then the IF/ID instruction will be stalled for one cycle (the ID stage of the implementation asserts stall_issue signal) to allow the load instruction to reach the EX/MEM latches. In the next cycle, the value read from the memory is forwarded to the ID stage. The implementation uses a simple “assume not taken” prediction strategy for jmp and beqz instructions. Consequently, when the IF/ID instruction turns out to be a taken branch (the ID stage of the implementation asserts the branch_taken signal), the pipeline squashes the subsequent instruction (the newly fetched one) and corrects the program counter. No instruction is fetched in a cycle in which no_op? is asserted.

The specification machine has the four observable components, that is, rf, dmem, pc and imem as its state components. Its transition function specifies the effect on these observable components of executing each of the six instructions atomically in one cycle.

4.1.2 Compositional construction of the abstraction function

This section is organized in a manner paralleling Section 3.1. Some notation which will be used throughout this chapter: q represents the current implementation state and i represents the current processor input. If rf is an implementation component, rf(q) stands for its value in state q, and will be referred to as just rf when the current implementation state is clear from the context. Similarly for a signal defined in the implementation machine (say branch_taken), we just write branch_taken to refer to branch_taken(q). The “primed” notation is used to refer to the value of a component after an implementation transition. So rf' will refer to rf(I_step(q,i)).
4.1.2.1 Unfinished instructions and the program order

The implementation can have four partially executed instructions at any time, one each in the four sets of pipeline latches shown in Figure 4.1. As explained in Chapter 3, an unfinished instruction is identified by the set of pipeline latches it resides in. Their program order is easily derived from the structure of the pipeline. So the four unfinished instructions in their program order are IF/ID (latest), ID/EX, EX/MEM and MEM/WB (earliest).

4.1.2.2 Specifying the completion functions

As an instruction gets executed in the pipeline, it passes through four phases corresponding to the four sets of pipeline latches. As explained in Chapter 3, the sets of pipeline latches is used to name these phases. A given unfinished instruction is in one of these phases at any given time. In fact, instruction IF/ID is in phase IF/ID and analogously for other instructions.

We now identify how to complete each of the four unfinished instructions and also identify if any of them can cause the subsequent instructions (in the program order) to be squashed. In order to do so, we need to identify how the information about a partially executed instruction is stored in the set of pipeline latches it resides in. Consider the instruction MEM/WB which is in phase MEM/WB. The intended purpose of the pipeline latches result_wb and dest_wb is that result_wb contains the value computed by the instruction MEM/WB to be written into the register (whose identifier is) contained in dest_wb, when dest_wb is not equal to zero_reg. So to complete MEM/WB, we need to write the value in result_wb into the register in dest_wb (when it is not equal to zero_reg). This is specified as Action_MEMWB, shown in [1]. The data memory, the program counter and the instruction memory are not affected by completing MEM/WB. The instruction MEM/WB will not cause the subsequent instructions to be squashed, so Squash_rest?_MEMWB always returns FALSE. Finally, Complete_MEMWB is specified following the template shown in Chapter 3 (the kill? argument is omitted as it is always FALSE).

```
Action_MEMWB(q:state_I): state_I = 
  q WITH [ (rf) := IF NOT (dest_wb(q) = zero_reg) THEN 
    assign(rf(q),dest_wb(q),result_wb(q)) 
  ELSE rf(q) ENDIF ]

Squash_rest?_MEMWB(q:state_I): bool = FALSE

Complete_MEMWB(q:state_I): state_I = Action_MEMWB(q)
```
Consider the instruction IF/ID, which is in phase IF/ID. The intended functionality of the IF stage is to fetch an instruction, place it in the pipeline latch instr_id and to increment pc. The bubble_id pipeline latch indicates whether the instruction IF/ID is valid or not. It might be invalid, for example, if no instruction was fetched in the previous cycle, say, due to no_op? being asserted. So in order to complete the execution of IF/ID, the completion function should do nothing if the instruction is not valid, otherwise it should update pc with the target address if it is a jmp or a taken beqz instruction, update dmem if it is a store instruction, and update rf if it is a load, alu_immed or alu_reg instruction according to the semantics of the instruction. The details of how these are done is obtained from the ISA level specification. We begin by writing four definitions: val_a and val_b specify the expected values for the two source operands, mem_addr specifying the memory address for the memory operations, and branch_taken_test specifying when the instruction is a taken branch. These definitions are shown in [2]. Note that we are not concerned with load dependency or data forwarding while specifying these definitions (or while specifying the completion function). It is assumed that the instructions ahead of IF/ID are completed and hence the source values can be directly read from the register file.

```plaintext
val_a(q : state_I) : value =
  IF src1(instr_id(q))=zero_reg THEN zero
  ELSE select(rf(q),src1(instr_id(q))) ENDIF

val_b(q : state_I) : value =
  IF op(instr_id(q))=alu_immed THEN short_immed_of(instr_id(q))
  ELSIF src2(instr_id(q))=zero_reg THEN zero
  ELSE select(rf(q),src2(instr_id(q))) ENDIF

mem_addr(q : state_I) : dmem_addr =
  value2dmem_addr(d_add(val_a(q),short_immed_of(instr_id(q))))

branch_taken_test(q : state_I) : bool =
  NOT bubble_id(q) AND (op(instr_id(q))=jmp OR
  (op(instr_id(q))=beqz AND val_a(q)=zero))
```

We then define Action_IF_ID specifying the desired effect on pc, dmem and rf of completing IF/ID. If this instruction turns out to be a taken branch, the subsequent instructions (incoming instruction in this example) are to be squashed. So Squash_rest?_IF_ID is defined to be just branch_taken_test. These definitions are shown in [3]. Similarly, the completion functions for other two unfinished instructions EX/MEM and ID/EX are
defined.

```plaintext
Action_IF_ID(q : state_I) : state_I =
q WITH [
  % Update pc with target address, (TA is the target address
  % computation function defined in the implementation.)
  (pc) := IF branch_taken_test(q) THEN TA(q)
  ELSE pc(q) ENDIF,

  % update dmem for a store instruction.
  (dmem) := IF NOT bubble_id(q) AND op(instr_id(q)) = store THEN
    write(dmem(q),mem_addr(q),val_b(q))
  ELSE dmem(q) ENDIF,

  % Update rf for load, alu_immed and alu_reg instructions.
  (rf) := IF NOT bubble_id(q) AND op(instr_id(q)) = load AND
    NOT (dest(instr_id(q)) = zero_reg) THEN
      assign(rf(q),dest(instr_id(q)),read(dmem(q),mem_addr(q)))
    ELSEIF NOT bubble_id(q) AND (op(instr_id(q))=alu_immed
      OR op(instr_id(q))=alu_reg) AND
      NOT (dest(instr_id(q))=zero_reg) THEN
        assign(rf(q),dest(instr_id(q)),alu(op(instr_id(q)),
          val_s(q),val_b(q)))
    ELSE rf(q) ENDIF ]

Squash_rest_IF_ID(q : state_I): bool = branch_taken_test(q)

% kill? argument is always FALSE.
Complete_IF_ID(q : state_I) : state_I = Action_IF_ID(q)
```

The completion functions for the unfinished instructions in the initial sets of pipeline latches are very close to the specification and it is very easy to derive them. (For example, Complete_IF_ID is almost the same as the ISA level specification). However the completion functions for the unfinished instructions in the later sets of pipeline latches are harder to derive as the user needs to understand how the information about the instruction is stored in the various pipeline latches but the functions themselves are usually much more compact. We believe that a typical designer has a very clear understanding of these and will not find the task of specifying the completion functions onerous.

### 4.1.2.3 Constructing the abstraction function

We begin by defining “Complete_till” functions to complete all the instructions till a given instruction in the program order (i.e., from the earliest till the given instruction in program order) as shown in [4]. We also define “Squash_rest_till” predicates computing the disjunction of the squashing predicates till a given instruction. Since the squashing predicates for the first three instructions are always FALSE in this example, we have only
one predicate \texttt{Squash\_rest?\_till\_IF\_ID} as shown in \ref{fig:example}. (The \texttt{kill?} argument is omitted for each of the four completion functions, since it is always \texttt{FALSE}).

\begin{verbatim}
| Complete\_till\_MEM\_WB(q;state\_I): state\_I = Complete\_MEM\_WB(q) |
| Complete\_till\_EX\_MEM(q;state\_I): state\_I = Complete\_EX\_MEM(Complete\_till\_MEM\_WB(q)) |
| Complete\_till\_ID\_EX(q;state\_I): state\_I = Complete\_ID\_EX(Complete\_till\_EX\_MEM(q)) |
| Complete\_till\_IF\_ID(q;state\_I): state\_I = Complete\_IF\_ID(Complete\_till\_ID\_EX(q)) |
| Squash\_rest?\_till\_IF\_ID(q;state\_I): bool = Squash\_rest?\_IF\_ID(Complete\_till\_IF\_ID\_EX(q)) |
\end{verbatim}

Then the abstraction function, which should have the cumulative effect of completing all the instructions, is defined as:

\begin{verbatim}
\texttt{ABS(q;state\_I): state\_A = projection(Complete\_till\_IF\_ID(q))}
\end{verbatim}

Observe that each completion function is defined assuming that the instructions ahead of it are already completed. This assumption is met in defining the abstraction function as above, since an instruction is completed in a context where the instructions ahead of it are already completed.

\subsection{Defining the synchronization function}

The synchronization function gives the number of instructions to be executed by the specification machine to keep it synchronized with the implementation machine (see Section 3.1.4). The synchronization function in this example returns zero under the following conditions:

- When \texttt{stall\_issue} is asserted in a cycle, the instruction IF/ID remains in the IF/ID latches, hence there is no room in the pipeline to fetch a new instruction.

- When \texttt{branch\_taken} is asserted in the current cycle or if the IF/ID instruction eventually turns out to be a taken branch (\texttt{Squash\_rest?\_till\_IF\_ID} is \texttt{TRUE}), then the new instruction ought not to be completed.

- When \texttt{no\_op?} is asserted and hence no new instruction is inserted into the pipeline.

Otherwise the synchronization function returns one. The definition is shown in \ref{fig:example}.

\begin{verbatim}
\texttt{sync\_fun(q;state\_I,i;inputs\_type): nat =}
  IF \texttt{stall\_issue(q) OR branch\_taken(q) OR}
  \texttt{Squash\_rest?\_till\_IF\_ID(q) OR no\_op?(i) THEN 0}
  ELSE 1 ENDIF
\end{verbatim}
4.1.3 Formulating the verification conditions

As explained in Chapter 3, the different verification conditions are generated by attempting to relate the values of the observable components, the squashing predicates and the source operands of the instructions in the present and the next implementation states.

Consider the instruction MEM/WD. In the current implementation machine transition, this instruction updates the register file and gets retired. So the contents of the register file in I_step(q,i) ought to be the same as after completing MEM/WD, that is, in state Complete_till_MEM_WB(q). (This is the first kind of verification condition for rf, discussed in Section 3.2.2, where one instruction, namely MEM/WD, is completed on both sides of the verification condition.) But dmem is updated in the MEM stage (and not in the WB stage which only writes the register file) and hence completing MEM/WD should have no effect on dmem. (See modification 2 described in Section 3.2.2.) pc is not to be affected by MEM/WD since the squashing predicate for MEM/WD is FALSE. imem is a read-only observable component and is not to be affected. (See modification 4 described in Section 3.2.2.) The verification conditions due to MEM/WD instruction are shown in 6.

\[
\begin{align*}
V_{C_{rf}}: & \text{CLAIM} \\
& rf(I_{\text{step}}(q,i)) = rf(\text{Complete}_{\text{till}}_{\text{MEM}}_{\text{WB}}(q)) \\
V_{C_{dmem}}: & \text{CLAIM} \\
& dmem(\text{Complete}_{\text{till}}_{\text{MEM}}_{\text{WB}}(q)) = dmem(q) \\
V_{C_{pc}}: & \text{CLAIM} \\
& pc(\text{Complete}_{\text{till}}_{\text{MEM}}_{\text{WB}}(q)) = pc(q) \\
V_{C_{imem}}: & \text{CLAIM} \\
& imem(\text{Complete}_{\text{till}}_{\text{MEM}}_{\text{WB}}(q)) = imem(q)
\end{align*}
\]

Now consider the effect of completing an additional instruction in the program order, that is, EX/MEM. This instruction is in phase EX/MEM in state q and goes to phase MEM/WD in I_{step}(q,i) (and hence resides in MEM/WD latches in I_{step}(q,i)). So Complete_MEM_WB can be used to complete this instruction in I_{step}(q,i). So the register file contents in the two states Complete_{till}_{EX}_{MEM}(q) and Complete_{till}_{MEM}_{WB}(I_{step}(q,i)) ought to be the same. Consider the effect on the data memory. EX/MEM will update dmem during the current implementation transition (if a store instruction) and Complete_MEM_WB has no effect on dmem in I_{step}(q,i). So the data memory contents

1Since q is an arbitrary reachable state, VC_{dmem} verification condition also applies to I_{step}(q,i),
in the two states Complete_till_EX_MEM(q) and Complete_till_MEM_WB(I_step(q,i)) ought to be equal too. pc and imem are not affected by completing EX/MEM. The verification conditions due to EX/MEM are shown in [7]. The verification conditions due to ID/EX follow the same pattern to those shown in [7].

```
VC2_rf: CLAIM
    rf(Complete_till_MEM_WB(I_step(q,i))) = rf(Complete_till_EX_MEM(q))

VC2_dmem: CLAIM
    dmem(Complete_till_MEM_WB(I_step(q,i))) = dmem(Complete_till_EX_MEM(q))

VC2_pc: CLAIM
    pc(Complete_till_EX_MEM(q)) = pc(q)

VC2_imem: CLAIM
    imem(Complete_till_EX_MEM(q)) = imem(q)
```

The verification conditions for IF/ID instruction are slightly different as the instruction may be stalled due to a load dependency on the instruction ID/EX. When the instruction IF/ID is not stalled (i.e., stall_issue is FALSE), it goes to ID/EX phase after an implementation transition and verification condition for rf is similar to the earlier ones (shown as VC4_rf_case1 in [8]). If it is stalled, then it remains in the IF/ID phase after an implementation transition (and hence in the IF/ID latches) and there is no valid instruction in ID/EX phase in I_step(q,i). Hence the register file contents ought to be equal when the instructions till IF/ID are completed in states q and I_step(q,i). This verification condition is shown as VC4_rf_case2 in [8]. The verification conditions for dmem due to the instruction IF/ID are exactly similar.

```
VC4_rf_case1: CLAIM
    NOT stall_issue(q) IMPLIES
    rf(Complete_till_ID_EX(I_step(q,i))) = rf(Complete_till_IF_ID(q))

VC4_rf_case2: CLAIM
    stall_issue(q) IMPLIES
    rf(Complete_till_IF_ID(I_step(q,i))) = rf(Complete_till_IF_ID(q))
```

Consider the truth value of Squash_rest? till IF_ID predicate. The implementation machine decides whether the IF/ID instruction is a taken branch or not in the ID stage. So when the IF/ID instruction is not stalled, the Squash_rest? till IF_ID predicate ought to have the same truth value as branch_taken signal. If IF/ID is stalled, then the

hence dmem(Complete_till_MEM_WB(I_step(q,i))) = dmem(I_step(q,i)).
truth value of the \texttt{Squash\_rest?\_till\_IF\_ID} predicate remains the same in states \(q\) and \(\text{l\_step}(q,i)\). These two verification conditions are shown in [9].

\begin{verbatim}
Squash\_rest?\_till\_IF\_ID\_VC1: \textbf{CLAIM}
  \texttt{(NOT stall\_issue(q)) \ IMPLIES}
  \texttt{(Squash\_rest?\_till\_IF\_ID(q) \iff branch\_taken(q))}

Squash\_rest?\_till\_IF\_ID\_VC2: \textbf{CLAIM}
  \texttt{stall\_issue(q) \ IMPLIES}
  \texttt{(Squash\_rest?\_till\_IF\_ID(\text{l\_step}(q,i)) \iff Squash\_rest?\_till\_IF\_ID(q))}
\end{verbatim}

The verification conditions for \(pc\) due to the IF/ID instruction are slightly involved as explained in Chapter 3. If none of the instructions till IF/ID (i.e., from the earliest till the IF/ID instruction) cause the rest of the instructions in the pipeline to be squashed, then completing them all ought not have any effect on \(pc\). This verification condition is shown as \texttt{VC4\_pc\_case1} in [10]. When IF/ID instruction is stalled, it remains in the IF/ID phase after an implementation transition, and implementation machine does not modify \(pc\) as there is no room in the pipeline to fetch a new instruction. So when IF/ID is completed in \(q\) and \(\text{l\_step}(q,i)\) (after the earlier instructions are completed), it ought to have the same effect on \(pc\). When IF/ID is not stalled, it goes to ID/EX phase after an implementation transition. In such a case, we can relate \(pc\) values when either \texttt{branch\_taken} is \texttt{TRUE} (the implementation machine modifies \(pc\) according to the branch target address), or when \texttt{branch\_taken} is \texttt{FALSE} but \texttt{no\_op?} is \texttt{TRUE} (no new instruction is fetched and hence \(pc\) is not updated on that count). These observations lead to three more verification conditions about \(pc\) as shown in [10].

\begin{verbatim}
VC4\_pc\_case1: \textbf{CLAIM}
  \texttt{NOT Squash\_rest?\_till\_IF\_ID(q) \ IMPLIES}
  \texttt{pc(Complete\_till\_IF\_ID(q)) = pc(q)}

VC4\_pc\_case2: \textbf{CLAIM}
  \texttt{stall\_issue(q) \ IMPLIES}
  \texttt{pc(Complete\_till\_IF\_ID(\text{l\_step}(q,i))) = pc(Complete\_till\_IF\_ID(q))}

VC4\_pc\_case3: \textbf{CLAIM}
  \texttt{(NOT stall\_issue(q) AND branch\_taken(q)) \ IMPLIES}
  \texttt{pc(Complete\_till\_ID\_EX(\text{l\_step}(q,i))) = pc(Complete\_till\_IF\_ID(q))}

VC4\_pc\_case4: \textbf{CLAIM}
  \texttt{(NOT stall\_issue(q) AND \texttt{NOT branch\_taken(q)} AND \texttt{no\_op?}(i)) \ IMPLIES}
  \texttt{pc(Complete\_till\_ID\_EX(\text{l\_step}(q,i))) = pc(Complete\_till\_IF\_ID(q))}
\end{verbatim}

The verification condition for \(imem\) due to IF/ID is similar to the earlier ones for \(imem\). Finally the verification conditions due to the new incoming instruction are as shown in
VCn_rf: CLAIM
   NOT (stall_issue(q) OR branch_taken(q) OR no_op?(i)) IMPLIES
   rf(Complete_till_IF_ID(I_step(q,i))) =
   rf(A_step(projection(Complete_till_IF_ID(q))))

VCn_dmem: CLAIM
   NOT (stall_issue(q) OR branch_taken(q) OR no_op?(i)) IMPLIES
   dmem(Complete_till_IF_ID(I_step(q,i))) =
   dmem(A_step(projection(Complete_till_IF_ID(q))))

VCn_pc: CLAIM
   NOT (stall_issue(q) OR branch_taken(q) OR no_op?(i)) IMPLIES
   pc(Complete_till_IF_ID(I_step(q,i))) =
   pc(A_step(projection(Complete_till_IF_ID(q))))

VCn_imem: CLAIM
   imem(A_step(projection(Complete_till_IF_ID(q)))) = imem(q)

The different verification conditions for rf and pc can be depicted pictorially as
shown in Figure 4.2. The rectangular boxes on the extreme left side of the figure denote
implementation states and the big circles on the extreme right side denote specification
states. The smaller circles in-between denote states obtained after instructions are
completed one at a time in their program order. Each verification condition asserts
that the contents of the observable component (rf or pc) in the two states pointed to by
it are the same. Some of the verification conditions are true only conditionally, but those
conditions are not shown in the figure to avoid clutter.

VC_next_operand_a: CLAIM
   (NOT stall_issue(q) AND NOT bubble_id(q)) IMPLIES
   next_operand_a(q) = val_a(Complete_till_ID_EX(q))

VC_next_operand_b: CLAIM
   (NOT stall_issue(q) AND NOT bubble_id(q)) IMPLIES
   next_operand_b(q) = val_b(Complete_till_ID_EX(q))

Finally, we come to the verification conditions expressing the correctness of the feed-
back logic. This correctness can be expressed succinctly in the completion functions
approach as discussed in Chapter 3. The IF/ID instruction obtains the two source
operands when it goes from the IF/ID phase to the ID/EX phase, either via the feedback
logic in case of dependencies or by directly reading from the register file (in case of no
dependencies). Intuitively, these values ought to be equal to the expected values for the
source operands, that is, values read from rf after completing the instructions ahead of
Figure 4.2 Verification conditions for rf and pc in the EX1.1 example
IF/ID. This observation leads to the verification conditions shown in [12].

Table 4.1 shows the summary of all the verification conditions generated in the overall proof decomposition.

4.1.4 Proof details

The first section here explains the proof of one of the verification conditions. The rest of the sections explain the general techniques/strategies used to prove all the verification conditions. The overall proof is organized into three parts:

1. Generating and proving certain rewrite rules, and defining some commonly used strategies.

2. Proving the verification conditions.

3. Proving the main proof obligations such as the commutativity and the projection obligations, and the invariants (if needed).

4.1.4.1 An example proof: proof of VC2_rf

The key idea while proving the verification conditions is to use the earlier proved verification conditions to contain the sizes of the terms generated and the number of cases examined.

- We begin by expanding the outermost function symbol on both sides of VC2_rf, that is, Complete_till_MEM_WB on the left hand side and Complete_till_EX_MEM on the

<table>
<thead>
<tr>
<th>Verification condition regarding</th>
<th>Number of verification conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>rf</td>
<td>6</td>
</tr>
<tr>
<td>dmem</td>
<td>6</td>
</tr>
<tr>
<td>pc</td>
<td>8</td>
</tr>
<tr>
<td>imem</td>
<td>5</td>
</tr>
<tr>
<td>Squash_rest?_till_IF_ID predicate</td>
<td>2</td>
</tr>
<tr>
<td>Correctness of the feedback logic</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.1. Verification conditions generated for EX1.1
right hand side. After this, we expand \texttt{Complete\_MEM\_WB} and \texttt{Action\_MEM\_WB} on the left hand side, and \texttt{Complete\_EX\_MEM} and \texttt{Action\_EX\_MEM} on the right hand side.

- On the left hand side of the resulting expression, terms of the following kind appear:
  
  L1: \( n\_mem\_wb(I\_step(q,i)) \) where \( n\_mem\_wb \) is a nonobservable pipeline latch from the MEM/WB set of pipeline latches.
  
  L2: \( rf(I\_step(q,i)) \)

- On the right hand side of the resulting expression, terms of the following kind appear:
  
  R1: \( n\_ex\_mem(\text{Complete\_till\_MEM\_WB}(q)) \) where \( n\_ex\_mem \) is a nonobservable pipeline latch from the EX/MEM set of pipeline latches.
  
  R2: \( rf(\text{Complete\_till\_MEM\_WB}(q)) \)
  
  R3: \( dmem(\text{Complete\_till\_MEM\_WB}(q)) \)

- The terms of kind “L1” can be rewritten into the “next” values for the corresponding nonobservable components using the implementation machine definitions. When these definitions are expanded, they will be referring to the nonobservable components in the EX/MEM set of pipeline latches and the observable \( dmem \). The terms of kind “R1” can again be rewritten into \( n\_ex\_mem(q) \) as the completion functions modify only the observable components, leaving all the nonobservable components unchanged. And “L2” can be rewritten into “R2” by \( VCl\_rf \), and “R3” rewritten into \( dmem(q) \) using \( VCl\_dmem \).

- After rewriting as described above, we check whether the resulting expression simplifies to \texttt{TRUE} by a brute force case-analysis. In PVS, by setting up the rewrites rules appropriately and defining suitable strategies, the above mentioned proof can be accomplished by issuing just three commands to the prover.

*Dealing with a failed proof:* If the proof does not go through, then one of the following must have happened:

\footnote{In describing the proofs in this dissertation, we refer to the “main” goal we are proving in a verification condition (\( rf(\ldots) = rf(\ldots) \) or \( dmem(\ldots) = dmem(\ldots) \) etc) as the consequent. When we just say the “left hand side” or the “right hand side” in a proof without qualifying it further, we are referring to this consequent.}
• The implementation has a design bug.

• The user made a mistake in the manual steps of the methodology.

• The user made a mistake in guiding the prover, although this is less likely since most of the verification conditions have similar and uniform proofs.

• The set of states considered in the proof of the commutativity obligation includes some unreachable states and needs to be restricted (by discovering an invariant property).

Our methodology does not give any direct assistance in pinpointing the reason for a failed proof. But we believe that by decomposing the proof into many verification conditions and reasoning about the pipelined design incrementally as our approach does, it is significantly easier to identify the cause for a failed proof and fix it.

4.1.4.2 Rewrite rules and proof strategies

As pointed out in the example proof, the term \texttt{dest\_mem(Complete\_till\_MEM\_WB(q))} is equal to \texttt{dest\_mem(q)} since completion functions are required not to modify the nonobservable components. We prove a lemma that captures this property and then instruct PVS to use the lemma as a rewrite rule. For each nonobservable component and for each unfinished instruction i, a lemma\(^3\) is generated that states that the nonobservable component is unaffected by completing the instructions till i. For example, one of the lemmas is shown in \footnote{13}. All these lemmas can be generated automatically once the completion functions are specified. Also, their proofs are all straightforward and exactly identical.

\begin{verbatim}
% Lemma for bubble_ex and Complete_till_EX_MEM used as a rewrite rule.
Rewrite_bubble_ex_Complete_till_EX_MEM: LEMMA
  FORALL (q : state_I):
    bubble_ex(Complete_till_EX_MEM(q)) = bubble_ex(q)
\end{verbatim}

Other lemmas used as rewrite rules include "record accessor" lemmas about the implementation and the specification machine states, and the lemmas about \texttt{select/assign} operations for the register file and \texttt{read/write} operations for the data memory (see Appendix B). In addition, the definitions from the implementation and the specification

\footnote{3It is possible to reduce the number of lemmas of this kind. For example, a lemma for \texttt{dest\_wb} and \texttt{Complete\_till\_IF\_ID} is unnecessary.}
machines, and the abstraction function related definitions are used as rewrite rules too. We then define strategies that let us instruct the prover to use these lemmas/definitions as rewrite rules and certain other simplification strategies. See Appendix C for more details. Most proofs need the implementation machine related and abstraction function related rewrite rules. So we define a PVS strategy \texttt{MY-REWRITES} to instruct the PVS prover to use these rewrite rules. This strategy also instructs PVS prover to use the earlier verification conditions as rewrite rules while proving a particular verification condition.

### 4.1.4.3 Proving the verification conditions

First consider the proofs of the verification conditions for the observable components. As pointed out in the earlier example proof, the idea is to expand the completion functions enough on both sides so as to be able to use the earlier verification conditions in simplifying the resulting expression. Consider proving \texttt{VC4_rf_case1}. We begin by issuing the commands (\texttt{SKOSIMP*}) and (\texttt{MY-REWRITES})—standard steps in all the proofs. We then expand the outermost "Complete\_fill" function on both sides. We then issue the command (\texttt{ASSERT}) to do the rewrites and the simplifications by decision procedures. This will cause a flurry of rewriting to take place on the both sides of the expression. On the left hand side, the functions \texttt{Complete\_ID\_EX} and \texttt{Action\_ID\_EX} are expanded, terms like \texttt{dest\_ex(Complete\_fill\_EX\_MEM(I\_step(q,i)))} are first rewritten to \texttt{dest\_ex(I\_step(q,i))} and then to \texttt{next\_dest\_ex(q)} and then expanded. The term \texttt{next\_operand\_a(q)} is rewritten using \texttt{VC\_next\_operand\_a}, the term \texttt{next\_operand\_b(q)} rewritten using \texttt{VC\_next\_operand\_b}, the term \texttt{rf(Complete\_fill\_EX\_MEM(I\_step(q,i)))} rewritten using \texttt{VC3\_rf} and the term \texttt{dmem(Complete\_fill\_EX\_MEM(I\_step(q,i)))} rewritten using \texttt{VC3\_dmem}. The left hand side of the resulting expression will contain terms referring to pipeline latches from IF/ID set of latches (i.e., \texttt{bubble\_id(q)} and \texttt{inst\_id(q)}), and the terms \texttt{rf(Complete\_fill\_ID\_EX(q))} and \texttt{dmem(Complete\_fill\_ID\_EX(q))}. On the right hand side, the functions \texttt{Complete\_IF\_ID} and \texttt{Action\_IF\_ID} are expanded, terms like \texttt{inst\_id(Complete\_fill\_ID\_EX(q))} are rewritten to \texttt{inst\_id(q)}. So the right hand side of the resulting expression will contain the same kind of terms as the left hand side. We now perform a brute-force case analysis on the resulting expression to check if it simplifies to \texttt{TRUE}. In a correct implementation, it ought simplify to \texttt{TRUE}. The sequence of commands that accomplish the above proof in PVS is shown in [14].
Consider proving VC4.pc_case4. The left hand side simplifies to pc(I_step(q,i)) using VC3.pc, and then to next_pc(q,i), and then to pc(q) (under those conditions shown in the lemma). Since stall_issue(q) and branch_taken(q) are FALSE, using Squash_rest?_till_IF_ID_VC1 and then VC4.pc_case1, the right hand simplifies to pc(q). There is no need to spell out this reasoning explicitly, the proof is accomplished by the three commands shown in 13.

The proof strategy for proving all other verification conditions for the observable components and the “Squash_rest?_till” predicates is similar to the ones presented above with minor differences. In all cases, the proofs are accomplished using no more than seven prover commands.

Now we consider proving the two verification conditions expressing the correctness of the feedback logic. Consider proving VC_next_operand.a. Its proof is carried out by considering the different cases for the source register being read (say r)—whether it is the same as the destination registers of the instructions in ID/EX, EX/MEM or MEM/WB latches or none of these cases. We consider these cases in the order shown above which helps to control the size of the expression generated when the completion functions are expanded. Consider the case when r is equal to the destination register of instruction ID/EX. Since Complete_ID_EX writes into this register, the expression on the right hand side simplifies to the value written by Complete_ID_EX. That the feedback logic indeed returns this value is then proved by case analysis on the resulting expression. Consider the case when r is the same as the destination of the instruction EX/MEM (and different from that of the destination of the instruction ID/EX). In this case, since Complete_ID_EX is writing a register different from r, its effect (and the terms due to it) can be ignored. Since r is the same as the register written by Complete_EX_MEM, the term on the right hand side simplifies to the value written by Complete_EX_MEM. That the feedback logic indeed returns this value is again proved by case analysis on the resulting expression.
The remaining cases are similarly handled. The proof of \( \text{VC}_{\text{next operand}, b} \) is similar to this proof.

### 4.1.4.4 Main proof obligations

We now consider proving the commutativity obligation for the different observable components. This is done by considering the different cases in the definition of the synchronization function. Consider the proof of commutativity obligation for \( \text{rf} \).

- Case when \( \text{stall} \_ \text{issue} \) is \( \text{TRUE} \): no new instruction is executed by the specification machine and by using \( \text{VC4}_{\text{rf} \_ \text{case2}} \), the proof is accomplished.

- Case when \( \text{stall} \_ \text{issue} \) is \( \text{FALSE} \) and \( \text{branch} \_ \text{taken} \) is \( \text{TRUE} \): no new instruction is executed by the specification machine and the new instruction fetched by the implementation is squashed. So expanding the definition of \( \text{Complete} \_ \text{till IF ID} \) on the implementation side and using \( \text{VC4}_{\text{rf} \_ \text{case1}} \), the proof is accomplished.

- Case when \( \text{stall} \_ \text{issue} \) and \( \text{branch} \_ \text{taken} \) are \( \text{FALSE} \) and \( \text{Squash} \_ \text{rest} \_ \text{till IF ID} \) is \( \text{TRUE} \): This case is ruled out by \( \text{Squash} \_ \text{rest} \_ \text{till IF ID} \_ \text{VC1} \).

- Case when \( \text{stall} \_ \text{issue} \) and \( \text{branch} \_ \text{taken} \) are \( \text{FALSE} \) and \( \text{no op} \) is \( \text{TRUE} \): no new instruction is executed by the specification machine and the implementation machine fetches no instruction. So expanding the definition of \( \text{Complete} \_ \text{till IF ID} \) on the implementation side and using \( \text{VC4}_{\text{rf} \_ \text{case1}} \), the proof is accomplished.

- Case when \( \text{stall} \_ \text{issue}, \text{branch} \_ \text{taken} \) and \( \text{no op} \) are all \( \text{FALSE} \): Follows immediately by \( \text{VCn}_{\text{rf}} \).

The proof of the commutativity obligation for \( \text{dmem} \) and \( \text{pc} \) are similar, and the proof for \( \text{imem} \) is straightforward.

Finally we consider the proof of the projection obligation. The processor is flushed when all the four pipeline latches do not contain any valid instruction. The \( \text{Flushed} \_ ? \) predicate as shown in [16]. The proof of the projection obligation is straightforward.

```plaintext
flushed?(q; state_l) : bool = bubble_id(q) AND bubble_ex(q) AND dest_ex(q) = zero_reg AND dest_mem(q) = zero_reg AND NOT store_flag(q) AND dest_vb(q) = zero_reg
```
**Invariant Properties:** One invariant property on the implementation states, discovered during the proof of VC3_rf, is required in this example. Its states that the dest_ex latch in the ID/EX set of latches is zero_reg whenever bubble_ex is TRUE, or whenever opcode_ex is equal to store or beqz or jmp. The proofs of the Inv_Init and Inv_Closure obligations are again straightforward.

### 4.1.5 Some observations

- The decomposition presented above is a particular decomposition that we chose. We could have avoided generating and proving, say VC2_rf, and proved that goal when it arises within the proof of VC3_rf if the prover can handle the term sizes. We also did not formulate any separate verification condition asserting the correctness of the source operands when the instruction IF/ID remained in the IF/ID phase due to being stalled (mentioned in Chapter 3), as its proof was easily accomplished in place in the verification conditions that used it. As pointed out in Chapter 3, our goal is to prove the commutativity obligation for the different observable components, and it is not mandatory that one formulate and prove all the verification conditions formulated above.

- The proof of a particular verification condition, say for rf, may use the verification conditions of other observable components too. For example, the proof of VC3_rf uses VC2_rf and VC2_dmem. Contrary to our presentation here, the actual verification proceeds incrementally—specify Complete_MEM_WB, formulate and prove VC1_rf, VC1_dmem, VC1_pc and VC1_imem, then consider the next unfinished instruction in EX_MEM latches, specify its completion function, formulate and prove the verification conditions due to it and so on.

### 4.1.6 Reducing the manual effort in specifying the completion functions

The completion functions are desired specifications for how an instruction in an intermediate state ought to complete, and the user needs to specify them. In some cases, it is possible to use certain heuristics to derive these definitions of the completion functions (and the squashing predicates) automatically from the implementation machine definition to reduce the manual effort. This technique is illustrated on EX1.1.
To derive the completion function for the instruction \texttt{MEM/WB}, we consider a (possibly) valid instruction in the \texttt{MEM/WB} latches and invalid instructions in the other three sets of latches. Referring to the definition of the \texttt{Flushed?} predicate in [16], this is achieved by assuming that the first five conjuncts are \texttt{TRUE} in the current implementation state. The \texttt{no_op?} input is assumed to be asserted. We then look at how the observable components are updated after a transition from this state, that is, in state \texttt{I.step(q,i)}. In this example, \texttt{dmem}, \texttt{pc} and \texttt{imem} are unchanged, and \texttt{rf} is updated according to \texttt{next_rf} definition. So the \texttt{Action_MEMWB} definition is taken to modify \texttt{rf} as above, and to leave other observable components unchanged. The \texttt{Action_MEMWB} definition thus derived is shown in [17].

```
Action_MEMWB(q : state_I) : state_I =
    q WITH [ (rf) := IF NOT(dest_wb(q)=zero_reg) THEN
            assign(rf(q),dest_wb(q),result_wb(q))
            ELSE rf(q) ENDIF ]
```

Now to derive the completion function for \texttt{EX/MEM}, we assume that a (possibly) valid instruction is present in \texttt{EX/MEM} latches, invalid instructions in other sets of latches, and that \texttt{no_op?} is asserted. Since the instruction \texttt{MEM/WB} updates \texttt{rf}, we use the \texttt{Action_MEMWB} definition in determining how the instruction \texttt{EX/MEM} updates \texttt{rf}. We replace the nonobservable components in the expression defining \texttt{rf} in \texttt{Action_MEMWB} (that is, \texttt{dest_wb} and \texttt{result_wb}) with their “\texttt{next_}” counterparts (that is, \texttt{next_dest_wb} and \texttt{next_result_wb}) and simplify the resulting expression. Also, the instruction \texttt{EX/MEM} will update \texttt{dmem} according to \texttt{next_dmem} definition, and leave \texttt{pc} and \texttt{imem} unchanged. So the \texttt{Action_EXMEM} definition is taken to modify \texttt{rf} and \texttt{dmem} as above, leaving \texttt{pc} and \texttt{imem} unchanged.

Similarly the definition of \texttt{Action_ID_EX} is derived as above using the expressions for \texttt{rf} and \texttt{dmem} from the definition of \texttt{Action_EXMEM}. Also, the definition of \texttt{Action_IF_ID} is derived in a similar fashion. The other details (like defining the “\texttt{Complete_till?”} functions and “\texttt{Squash_rest?} till’” predicates) not pointed out here are similar to those in Section 4.1.2. The proof decomposition is exactly identical as in Section 4.1.3. The proof details are similar but with an important difference: while the proof in Section 4.1.4 needed an invariant property, this proof required none.

This way of deriving the completion functions from the implementation machine reduces the manual effort, but has the disadvantage that the implementation machine
is being verified against itself. This contradicts our view of these as desired specifications and negates our goal of incremental verification. In the example above, a design bug in the WE stage would go undetected and appear in the completion functions that are being built up. All design bugs will eventually be caught, however, since the commutativity obligation uses the "correct" specification provided by the user instead of being generated from the implementation. To combine the advantages of both, a hybrid approach can be used where explicitly provided and symbolically generated completion functions are used in combination. For example, the completion function could be derived for the last stage, specified for the penultimate stage, and then derived for the stage before that (from the specification for the penultimate stage) and so on. This line of research is outside the scope of this dissertation.

4.2 Application to EX1.2

4.2.1 Processor details

EX1.2, shown in Figure 4.3, is a dual issue implementation (similar to the design verified in [12]) of the same subset of the DLX processor considered in Section 4.1. Both the pipelines have similar structure as Figure 4.1 except that the second pipeline only executes alu-immed and alu-reg instructions. In addition, the processor has one instruction buffer location.

The issue logic in this implementation is fairly involved, issuing zero to two instructions per cycle. Instruction IF/ID.1 can get stalled for a cycle due to a load dependency on the instruction ID/EX.1 as in the EX1.1 example (the implementation asserts stall_issue_1 signal). Instruction IF/ID.2 can get stalled either because IF/ID.1 is stalled (there is no out-of-order issue here) or due to a load dependency on the instruction ID/EX.1 or due to a dependency on instruction IF/ID.1 or if it is neither an alu-immediate nor an alu-reg instruction (the implementation asserts stall_issue_2 signal). The implementation keeps the valid instructions in three IF/ID latches "packed", that is, when IF/ID.1 instruction is invalid, the later two instructions are invalid too, and when IF/ID.2 is invalid, IF/ID.3 is invalid too. So if IF/ID.1 is issued and IF/ID.2 is stalled,

\[\text{In fact, if the heuristics are suitably formalized, then some of the verification conditions can be shown to hold by construction and it is not necessary to prove them.}\]

\[\text{5An instruction is stalled only when it is valid. So it follows that if stall_issue_1 is TRUE and stall_issue_2 is FALSE, then IF/ID.2 is invalid. This is a minor observation, but will be used in the discussions later.}\]
then during that cycle, IF/ID_2 instruction moves to the IF/ID_1 latches. The branch outcome is decided in the ID stage of the first pipeline, and if IF/ID_1 is a taken branch (the implementation asserts branch_taken signal), then instructions IF/ID_2 and IF/ID_3 are squashed, and in the next cycle, all three IF/ID latches do not contain any valid instruction. The fetch logic fetches new instructions in pairs whenever there is enough room in the processor. For branch instructions, it uses a “assume not taken” strategy and continues instruction fetching speculatively. These factors lead to a wide variation in the latency of an instruction waiting to be issued. Once instructions are issued, they flow down the pipeline and complete execution as in the EX1.1 example.

4.2.2 Completion functions and the abstraction function

Only those details that are different from that of the EX1.1 example are illustrated. There can be nine unfinished instructions in the pipeline, so there are nine completion functions to be specified. Their program order is easily derived from the structure of the pipeline: MEM/WB_1 (earliest), MEM/WB_2, EX/MEM_1, EX/MEM_2, ID/EX_1, ID/EX_2, IF/ID_1,
IF/ID\_2 IF/ID\_3 (latest). The instructions and the different phases they can be in are named as before.

The specification of the “Action” functions for all the unfinished instructions is similar as in the EX1.1 example, so only the specification of the squashing predicates is discussed. The squashing predicates are always FALSE for the first six instructions. IF/ID\_1 can cause the rest of the instructions to be squashed if it eventually turns out to be a taken branch. branch\_taken\_test\_1 predicate shown in tests if IF/ID\_1 is a taken branch or not; it is a taken branch when it is valid and either it is a jmp instruction or it is a beqz instruction and the first source operand is zero. The definitions of Squash\_rest? IF/ID\_1 and Complete IF/ID\_1 are then as shown in 18. The predicates branch\_taken\_test\_2 and branch\_taken\_test\_3 are similarly defined, as IF/ID\_2 and IF/ID\_3 could potentially be taken branches. These definitions, and the squashing predicates and completion functions for IF/ID\_2 and IF/ID\_3 are shown in 18. Observe that these two instructions may be “killed” by an earlier instruction.

```plaintext
%% val_operand_a_1 is the value of the first source operand.
%% bubble_id_1 and instr_id_1 have similar meaning as in EX1.1
branch_taken_test_1(q : state_1) : bool =
   NOT bubble_id_1(q) AND (op(instr_id_1(q)) = jmp OR
   (op(instr_id_1(q)) = beqz AND val_operand_a_1(q) = zero))

Squash_rest? IF/ID\_1(q : state_1) : bool = branch_taken_test_1(q)

%% kill? argument is FALSE.
Complete_IF/ID\_1(q : state_1): state_1 = Action_IF/ID\_1(q)

%% For IF/ID\_2 instruction.
branch_taken_test_2(q : state_1) : bool =
   NOT bubble_id_2(q) AND (op(instr_id_2(q)) = jmp OR
   (op(instr_id_2(q)) = beqz AND val_operand_a_2(q) = zero))

Squash_rest? IF/ID\_2(q : state_1) : bool = branch_taken_test_2(q)

Complete_IF/ID\_2(q : state_1,kill?): bool) : state_1 =
   IF kill? THEN q
   ELSE Action_IF/ID\_2(q) ENDIF

%% For IF/ID\_3 instruction.
branch_taken_test_3(q : state_1) : bool =
   NOT bubble_id_3(q) AND (op(instr_id_3(q)) = jmp OR
   (op(instr_id_3(q)) = beqz AND val_operand_a_3(q) = zero))

Squash_rest? IF/ID\_3(q : state_1) : bool = branch_taken_test_3(q)

Complete_IF/ID\_3(q : state_1,kill?): bool) : state_1 =
   IF kill? THEN q
   ELSE Action_IF/ID\_3(q) ENDIF
```
We then define the “Complete_till” functions and the “Squash_rest?_till” predicates for all the unfinished instructions. These definitions for the last three instructions are shown in [19]. Recall that the “Squash_rest?_till” predicates are defined as the disjunction of the squashing predicates of all the unfinished instructions till that given instruction. Also observe the kill? arguments passed to Complete_IF_ID_2 and Complete_IF_ID_3 in the definitions of Complete_till_IF_ID_2 and Complete_till_IF_ID_3. Instruction IF/ID_3 is to be killed when any of the instructions ahead of it turns out to be a taken branch, that is, in this example, when Squash_rest?_till_IF_ID_2 evaluates to TRUE. Finally, the abstraction function definition is as shown in [19]. Note how the completion functions and the squashing predicates are applied to an implementation state where the instructions ahead of the one under consideration are all completed.

```plaintext
Complete_till_IF_ID_1(q: state_I): state_I = Complete_IF_ID_1(Complete_till_ID_EX_2(q))

Squash_rest?_till_IF_ID_1(q: state_I): bool = Squash_rest?_IF_ID_1(Complete_till_ID_EX_2(q))

Complete_till_IF_ID_2(q: state_I): state_I = Complete_IF_ID_2(Complete_till_IF_ID_1(q), Squash_rest?_till_IF_ID_1(q))

Squash_rest?_till_IF_ID_2(q: state_I): bool = Squash_rest?_till_IF_ID_1(q) OR Squash_rest?_IF_ID_1(Complete_till_IF_ID_1(q))

Complete_till_IF_ID_3(q: state_I): state_I = Complete_IF_ID_3(Complete_till_IF_ID_2(q), Squash_rest?_till_IF_ID_2(q))

Squash_rest?_till_IF_ID_3(q: state_I): bool = Squash_rest?_till_IF_ID_2(q) OR Squash_rest?_IF_ID_3(Complete_till_IF_ID_2(q))

ABS(q: state_I): state_A = projection(Complete_till_IF_ID_3(q))
```

The synchronization function definition is shown in [20]. No new instruction is executed by the specification machine when branch_taken is asserted or when any of the instructions in the IF/ID latches turn out to be a taken branch or if there is no room to fetch a new instruction (remains_123_three and remains_123_two signals mean that either three or two instructions remain in the IF/ID latches) or if no_pp? is asserted. An interesting case is when two instructions are fetched and the first one turns out to be a taken branch. In that case, only one instruction is to be executed by the specification
machine. This condition is specified using `spec_branch_taken_test_1` predicate shown in [20].

```plaintext
spec_branch_taken_test_1(as: state_A) : bool =
  op(instr(as)) = jmp OR
  ((op(instr(as))) = beqz AND srcidata(as) = zero)

sync_fun(q: state_I, i: inputs_type) : nat =
  IF branch_taken(q) OR Squash_rest?_till_IF_ID_1(q) THEN 0
  ELSIF Squash_rest?_till_IF_ID_2(q) THEN 0
  ELSIF remains_123_three(q) OR remains_123_two(q) THEN 0
  ELSIF Squash_rest?_till_IF_ID_3(q) THEN 0
  ELSIF no_op?(i) THEN 0
  ELSIF spec_branch_taken_test_1(projection(Complete_till_IF_ID_3(q))) THEN 1
  ELSE 2 ENDIF
```

### 4.2.3 Decomposition and the proof details

We illustrate the verification conditions due to IF/ID.2 and the later instructions, pointing out many of the details. The verification conditions due to other instructions are similar to these or those from EX1.1 example, and we point out some interesting differences only.

The verification conditions for rf due to IF/ID.1 and IF/ID.2 are shown in Figure 4.4. (Again, the preconditions of these verification conditions are not shown to avoid clutter. Also, Complete_IF_ID.2 has a kill? argument which is shown in [19] not explicitly shown in the figure here.) Consider how instruction IF/ID.2 changes phase during an implementation transition. There are four possible scenarios, and the verification conditions in these scenarios build on the verification conditions for the earlier instruction (VC7_rf_case1 and VC7_rf_case2, also shown in the figure).

- **stall_issue.1** and **stall_issue.2** are both FALSE: Instruction IF/ID.2 changes phase to ID/EX.2 by getting issued in the second pipeline. The verification condition in this case is VC8_rf_case1.

- **stall_issue.1** is FALSE and **stall_issue.2** is TRUE: Since IF/ID.1 is issued, it "vacates" the IF/ID.1 latches. So instruction IF/ID.2 changes phase to IF/ID.1 by moving to IF/ID.1 latches after an implementation transition. And there is no valid instruction in ID/EX.2 phase in state I_step(q,i). The verification condition in this case is VC8_rf_case2.
Earlier verification conditions

Verification conditions for rf due to IF/ID 1 and IF/ID 2

Figure 4.4.
• \textbf{stall\_issue\_1} and \textbf{stall\_issue\_2} are both TRUE: IF/ID\_2 remains in the IF/ID\_2 phase and the verification condition in this case is \textit{VC8\_rf\_case3}.

• \textbf{stall\_issue\_1} is TRUE and \textbf{stall\_issue\_2} is FALSE: From the observation in Section 4.2.1, it follows that IF/ID\_2 is invalid and the verification condition in this case is \textit{VC8\_rf\_case4}.

Now consider proving these verification conditions. \textit{VC8\_rf\_case2} is shown in [21].

When either \texttt{branch\_taken} or \texttt{Squash\_rest?\_till\_IF\_ID\_1} are TRUE, it is possible to accomplish the proof of the commutativity obligation using the earlier verification conditions, hence we may assume both these conditions are FALSE in the subsequent verification conditions. Since \textbf{stall\_issue\_1} is FALSE, we expand the “Complete\_till” functions enough on both sides so that we can use the verification condition of IF/ID\_1 for the case when \textbf{stall\_issue\_1} is FALSE, which is \textit{VC7\_rf\_case1}. So we expand \texttt{Complete\_till\_IF\_ID\_1} and then \texttt{Complete\_till\_ID\_EX\_2} on the left hand side, and \texttt{Complete\_till\_IF\_ID\_2} on the right hand side. Since \texttt{Squash\_rest?\_till\_IF\_ID\_1} is FALSE, the \texttt{kill?} argument to \texttt{Complete\_IF\_ID\_2} is FALSE. Issuing the command (\texttt{ASSERT}) does the necessary definition expansions, rewriting, uses \textit{VC7\_rf\_case1} and accomplishes the proof. The proofs of other three verification conditions are equally straightforward.

\begin{verbatim}
VC8_rf_case2: CLAIM
(NOT branch_taken(q) AND NOT Squash_rest?_till_IF_ID_1(q) AND
 NOT stall_issue_1(q) AND stall_issue_2(q)) IMPLIES
 rf(Complete_till_IF_ID_1(I_step(q,i))) = rf(Complete_till_IF_ID_2(q))
\end{verbatim}

Now consider the verification conditions due to \texttt{Squash\_rest?\_till\_IF\_ID\_2}. These three verification conditions are shown in [22]. When \textbf{stall\_issue\_2} is FALSE, IF/ID\_2 can not be a \texttt{jmp} or a \texttt{beqz} instruction, hence \texttt{Squash\_rest?\_IF\_ID\_2} is FALSE, and hence the verification condition \texttt{Squash\_rest?\_till\_IF\_ID\_2\_VC1}. So assume \textbf{stall\_issue\_2} is TRUE. If \textbf{stall\_issue\_1} is FALSE, then as pointed out before, IF/ID\_2 goes to phase IF/ID\_1 after an implementation transition, IF/ID\_2 will cause the rest of the instructions to be squashed in state \texttt{q} if and only if IF/ID\_1 causes the rest of the instructions to be squashed in state \texttt{I\_step(q,i)}, and hence \texttt{Squash\_rest?\_till\_IF\_ID\_2\_VC2}. The explanation for the third verification condition is similar. Their proofs are again straightforward.
Squash_rest?_till_IF_ID_2_VC1: CLAIM
(NOT stall_issue_2(q) AND NOT Squash_rest?_till_IF_ID_1(q)) IMPLIES
(NOT Squash_rest?_till_IF_ID_2(q))

Squash_rest?_till_IF_ID_2_VC2: CLAIM
(NOT stall_issue_1(q) AND stall_issue_2(q) AND NOT branch_taken(q) AND NOT Squash_rest?_till_IF_ID_1(q)) IMPLIES
(Squash_rest?_till_IF_ID_1(I_step(q,i)) IFF Squash_rest?_till_IF_ID_2(q))

Squash_rest?_till_IF_ID_2_VC3: CLAIM
(stall_issue_1(q) AND stall_issue_2(q) AND NOT branch_taken(q) AND NOT Squash_rest?_till_IF_ID_1(q)) IMPLIES
(Squash_rest?_till_IF_ID_2(I_step(q,i)) IFF Squash_rest?_till_IF_ID_2(q))

VC8_pc_case1: CLAIM
(NOT Squash_rest?_till_IF_ID_2(q) IMPLIES
pc(Complete_till_IF_ID_2(q)) = pc(q)

VC8_pc_case2: CLAIM
(NOT stall_issue_1(q) AND NOT stall_issue_2(q) AND NOT branch_taken(q) AND NOT Squash_rest?_till_IF_ID_1(q)
AND no_op?(i)) IMPLIES
pc(Complete_till_ID_EX_2(I_step(q,i))) = pc(Complete_till_IF_ID_2(q))

VC8_pc_case6: CLAIM
(NOT stall_issue_1(q) AND stall_issue_2(q) AND Squash_rest?_till_IF_ID_2(q) AND NOT branch_taken(q)
AND NOT Squash_rest?_till_IF_ID_1(q)) IMPLIES
pc(Complete_till_IF_ID_1(I_step(q,i))) = pc(Complete_till_IF_ID_2(q))

Consider the verification conditions for pc, which are slightly involved since pc is updated at two places in the pipeline. Some of these verification conditions are shown in [23]. VC8_pc_case1 is similar to VC4_pc_case1 in EX1.1 example shown in [10]. The implementation machine modifies pc when either branch_taken is TRUE or when new instructions are fetched. As pointed out earlier, while formulating the verification conditions for IF/ID_2, we assume that branch_taken is FALSE. And we relate the effect of completing IF/ID_2 on pc in the present and the next implementation states under the conditions when the pipeline fetches no new instructions. (If the pipeline fetched a new instruction and incremented pc, we are yet to consider that instruction in our scheme of looking at the instructions, that is, the program order. Refer to Chapter 3 for details.) There are four scenarios here depending on the truth values of stall_issue_1 and stall_issue_2. When both are FALSE, then as before IF/ID_2 changes phase to ID/EX_2, and the verification condition is as shown as VC8_pc_case2 in [23]. The verification conditions for the other three scenarios are similar to their counterparts for
rf. Finally, when IF/ID\_2 causes the rest of the instructions to be squashed, it should cause them to be squashed after an implementation transition too and the final effect on the pc ought to be same in states q and I\_step(q,i). This verification condition is shown as VC8\_pc\_case6. The proofs of these verification conditions are all similar to the earlier described proofs.

Now when Squash\_rest?\_till\_IF/ID\_2 is TRUE, we can accomplish the proof of the commutativity obligation without considering the verification conditions due to subsequent instructions. There are two cases depending on whether stall\_issue\_1 is TRUE or not. Consider the case when stall\_issue\_1 is TRUE, which is shown in \[24\]. Since Squash\_rest?\_till\_IF/ID\_2 is TRUE and Squash\_rest?\_till\_IF/ID\_1 is FALSE, by Squash\_rest?\_till\_IF/ID\_2\_VC1, it follows that stall\_issue\_2 is TRUE, and hence using Squash\_rest?\_till\_IF/ID\_2\_VC3, it follows that Squash\_rest?\_till\_IF/ID\_2 is TRUE in state I\_step(q,i). So the kill? argument for Complete\_IF/ID\_3 is TRUE on both sides, and using VC8\_rf\_case3, the proof is accomplished. The case when stall\_issue\_1 is FALSE is similar, except that using Squash\_rest?\_till\_IF/ID\_2\_VC2, we conclude that kill? argument to Complete\_IF/ID\_2 is TRUE too on the left hand side, and then using VC8\_rf\_case2, the proof is accomplished.

\[
\text{commutes\_rf\_case3: LEMMA} \\
\text{(NOT branch\_taken(q) AND NOT Squash\_rest?\_till\_IF/ID\_1(q)}} \\
\text{AND Squash\_rest?\_till\_IF/ID\_2(q) AND stall\_issue\_1(q)) IMPLIES} \\
\text{rf(ABS(I\_step(q,i)))) = rf(A\_step\_new(ABS(q),0))}
\]

We skip the details of the verification conditions due to IF/ID\_3 and consider those due to the first incoming instruction i1. There are four cases:

- None of the instructions in the IF/ID latches remain there after an implementation transition. This happens when both stall\_issue\_1 and stall\_issue\_2 are FALSE and bubble\_id\_3 is TRUE. i1 will be in phase IF/ID\_1 after an implementation transition in such case.

- One of the instructions in the IF/ID latches remains there after an implementation transition. This can happen either when both the instructions, IF/ID\_1 and IF/ID\_2, are not stalled and IF/ID\_3 is a valid instruction or when one of the instructions, IF/ID\_1 or IF/ID\_2, is stalled and IF/ID\_3 is not a valid instruction. These conditions are:
1. stall_issue_1 and stall_issue_2 are FALSE and bubble_id_3 is FALSE.
2. stall_issue_1 is FALSE, stall_issue_2 is TRUE, and bubble_id_3 is TRUE.
3. stall_issue_1 is TRUE, stall_issue_2 is FALSE, and bubble_id_3 is TRUE.

\( i_{11} \) will in in IF/ID.2 phase after an implementation transition in these cases.

\begin{verbatim}
VCi1_rf_case1: CLAIM
  (NOT stall_issue_1(q) AND NOT stall_issue_2(q) AND bubble_id_3(q) AND
   NOT branch_taken(q) AND NOT Squash_rest7_till_IF_ID_1(q) AND
   NOT Squash_rest7_till_IF_ID_2(q) AND NOT Squash_rest7_till_IF_ID_3(q)
   AND NOT no_op?(i)) IMPLIES
   rf(Complete_till_IF_ID_1(I_step(q,i))) =
   rf(A_step(projection(Complete_till_IF_ID_3(q))))

spec_branch_taken_test_1_VC1: CLAIM
  (NOT stall_issue_1(q) AND NOT stall_issue_2(q) AND bubble_id_3(q) AND
   NOT branch_taken(q) AND NOT Squash_rest7_till_IF_ID_1(q) AND
   NOT Squash_rest7_till_IF_ID_2(q) AND NOT Squash_rest7_till_IF_ID_3(q)
   AND NOT no_op?(i)) IMPLIES
  (Squash_rest7_till_IF_ID_1(I_step(q,i)) \iff
   spec_branch_taken_test_1(projection(Complete_till_IF_ID_3(q))))
\end{verbatim}

The verification conditions for \( rf \) and \( \text{spec\_branch\_taken\_test\_1} \) predicate in the first of these four cases are shown in [25]. The other three verification conditions are similarly formulated and their proofs are all straightforward. Now, in all the four scenarios, if \( i_{11} \) turns out to be a taken branch it is possible to accomplish the proof of the commutativity obligation. Consider again the first of these four cases. The synchronization function returns one in such a case, and by \( \text{spec\_branch\_taken\_test\_1\_VC1} \), it follows that \( \text{Squash\_rest7\_till\_IF\_ID\_1(I\_step(q,i))} \) is TRUE, and hence the \( \text{kill?} \) argument to both \( \text{Complete\_IF\_ID\_3} \) and \( \text{Complete\_IF\_ID\_2} \) are TRUE on the implementation side, and hence by using \( \text{VCi1\_rf\_case1} \), the proof is accomplished. Now if \( i_{11} \) does not turn out to be a taken branch, then the effect of the second incoming instruction needs to be considered in each of these four cases, which is not discussed here.

The total number of verification conditions in the overall proof decomposition is shown in Table 4.2. The rather large number of verification conditions is mainly due to the wide variation in the latency of an instruction waiting to be issued. (It is possible to decrease the number of verification conditions by combining many similar verification conditions into one, but this will decrease the uniformity of the proofs.) We make two observations regarding the verification conditions due to the first seven instructions.
Table 4.2. Verification conditions generated for EX1.2

<table>
<thead>
<tr>
<th>Verification conditions regarding</th>
<th>The number of verification conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>rf</td>
<td>26</td>
</tr>
<tr>
<td>dmem</td>
<td>26</td>
</tr>
<tr>
<td>pc</td>
<td>33</td>
</tr>
<tr>
<td>imem</td>
<td>9</td>
</tr>
<tr>
<td>“Squash_rest_till” predicates</td>
<td>11</td>
</tr>
<tr>
<td>Correctness of the feedback logic</td>
<td>6</td>
</tr>
</tbody>
</table>

- Since the pipeline retires two instructions in an implementation transition, it is not possible to consider the effect of the first instruction (i.e., MEM/WB_1) only; the first verification condition for rf we have relates the register file contents in state I_step(q,i) and in a state where both MEM/WB_1 AND MEM/WB_2 are completed.

- The verification condition for IF/ID_1 when stall_issue_1 is TRUE is shown as VC7_rf_case2 in Figure 4.4. Its proof uses an earlier verification condition that relates the two states s1 and s2 in the figure. Since there are three completion functions on the implementation side, we found it convenient to introduce an auxiliary verification condition VC7_rf_case2_aux as shown in the Figure 4.4, and then prove VC7_rf_case2 using VC7_rf_case2_aux. There are two auxiliary verification conditions on the correctness of the feedback logic too, and the numbers shown in Table 4.2 include these.

The proof strategies used are similar to the EX1.1 example. There are some “pc arithmetic” axioms, and simple properties of stall_issue_1, stall_issue_2 and branch_taken signals used as rewrite rules. None of the verification conditions, other than those corresponding to the feedback logic, need more than eight prover commands. The proofs of the verification conditions due to the feedback logic proceed by considering the different cases for the source register being read, as in the EX1.1 example. The required invariant properties and the proofs of the Inv_Init and the Inv_Closure obligations are straightforward.
CHAPTER 5
PROCESSORS WITH OUT-OF-ORDER EXECUTION AND IN-ORDER COMPLETION

This chapter discusses the verification of one Class 2 example, that is, with out-of-order execution and in-order completion. The example EX2.2 is a substantially involved design with a reorder buffer, a store buffer, speculative execution and precise exceptions. The different steps in applying the completion functions approach to this example are discussed.

5.1 Details of EX2.2

At the specification level, the state of the processor is represented by a register file rf, a special register file sp_rf accessed only by privileged/special instructions, a data memory dmem, a mode flag mode_flag, a program counter pc and an instruction memory imem. The processor can operate in either user or supervisory mode, and the current mode is maintained in mode_flag. Four instructions are supported in user mode: alu instruction for performing arithmetic and logical operations, load and store instructions for accessing the data memory, and beq instruction for performing conditional branches. (Their semantics are as explained in Appendix B.) In supervisory mode, three additional privileged instructions are allowed: rfeh instruction for returning from an exception handler, mfsr and msr instructions for moving data from and to the special register file. Three types of exceptions are possible—arithmetic exception raised by an alu instruction, data access exception raised by load and store instructions when the memory address is outside legal bounds (two special registers LOW_BOUND and HIGH_BOUND maintain the legal bounds, and this is checked only in user mode), and

\[1\text{The verification of EX2.1, another example from this class, is not described in this dissertation since it has only a subset of the features of EX2.2.}\]
an illegal instruction exception (raised either when an illegal opcode is encountered or when a privileged instruction is executed in user mode). When an exception is raised, the processor saves the address of the faulting instruction in a special register RESTART and jumps to an exception handler (depending on the exception type) and changes to supervisory mode. After processing a raised exception, the processor can return to user mode by executing rfeh instruction, which also restarts the instruction execution at the address in RESTART register.

An implementation of this processor is shown in Figure 5.1. A reorder buffer is used to maintain the program order of the executing instructions so that they can be committed in that order to respect the ISA semantics. Every executing instruction has a reorder buffer entry associated with it, and each such entry maintains some information about the executing instruction such as its destination register, whether or not it has raised an exception etc. The reorder buffer is implemented as a circular fifo queue with its tail rb_tail pointing to the earliest issued instruction, head rb_head pointing to the first free location in the buffer, rb_count maintaining the number of executing instructions in the buffer, and its size being an uninterpreted positive constant RB. Register translation tables (regular and special), with one entry per register, provide the reorder buffer index of the latest pending instruction writing a particular register. An entry of zero means that register is not being written by any of the executing instructions. “Alu/Branch/Special Instr. Unit” (referred to as ABS Unit) executes alu, beq and all the special instructions. The (unordered pool of) reservation stations in the ABS Unit hold the instructions sent to this unit until they are ready to be dispatched onto an appropriate execution unit (contained within the ABS Unit). These instructions are executed in an out-of-order manner by the multiple execution units present in the ABS Unit. The number of reservation stations and the execution units in the ABS Unit is given by uninterpreted constants ABS_rs and ABS_en. Instructions load and store are executed by the “Load Store Unit” (referred to as the LS Unit) where the reservation stations form a circular fifo queue storing the instructions in their program order. Again, lsu_rs_tail points to the earliest instruction, lsu_rs_head points to the first free reservation station, lsu_rs_count maintains the number of occupied reservation

\(^2\)The reorder buffer indices and other indices to be introduced later (execution unit indices, store buffer indices, reservation station indices etc) are all modeled as natural numbers, ranging from one till a maximum value (an uninterpreted constant).
stations, and the total number of reservation stations is given by an uninterpreted constant \( \text{LSU}_{rs} \). These instructions are executed in their program order by the single execution unit present in the LS Unit. Each reservation station (in both ABS Unit and LS Unit) holds some information about an instruction such as its source values, the reorder buffer index of the instructions producing its source values, etc. The store buffer has one entry per \texttt{store} instruction (if it has not raised any exceptions) recording the memory address and the value to be stored. Each such instruction is later committed to the data memory if it is not squashed by any of the earlier instructions. The store buffer is again implemented as a circular fifo queue, with head and tail pointers, keeping the instructions to be written to the data memory in their program order. A scheduler controls the movement of the instructions through the execution pipeline (such as being

\textbf{Figure 5.1.} Block diagram model of our implementation of EX2.2
dispatched, executed, etc.), and its behavior is modeled by a collection of axioms that subsume a whole range of implementation possibilities. Instructions are fetched from the instruction memory using the program counter; and the implementation also takes a no_op? input, which suppresses an instruction fetch when asserted.

An instruction is issued (the implementation asserts ifu_instr_issued? signal) by allocating an entry for it at the head of the reorder buffer and (depending on the instruction type) either a free reservation station (sch_new_slot) in the ABS Unit or a free reservation station at the head of the queue of reservation stations in the LS Unit. If the instruction being issued is an illegal instruction, then that fact is recorded in the reorder buffer entry and such an instruction is not issued to either of the units. If the instruction being issued is a branch instruction, then the program counter is modified according to a "predicted" branch target address (sch_pred_target, an unconstrained arbitrary value), and in the next cycle the new instruction is fetched from this address. No instruction is issued if there is no free reorder buffer location or if there are no free reservation stations or if no_op? is asserted or if the processor is being restarted (for reasons detailed later). The reorder buffer entry for the instruction being issued is appropriately initialized and the RTT entry corresponding to the destination of the instruction is updated to reflect the fact that the instruction being issued is the latest one to write that register. If the source registers are not being written by previously issued pending instructions (checked using the RTT) then their values are obtained from the register file, otherwise the reorder buffer indices of the instructions providing the source operands are maintained in the reservation station. Issued instructions wait for their source operands to become ready, monitoring all the execution units if they produce the values they are waiting for. An instruction can be dispatched when its source operands are ready and a free execution unit is available. In the ABS Unit, the instructions can be dispatched in an out-of-order manner. The sch_abs_di? and sch_abs_di outputs of the scheduler (each a ABS_eu-wide vector) determine whether or not to dispatch an instruction to a particular execution unit and the reservation station from where to dispatch. 

However, in the LS Unit, only the instruction at the tail of the queue of reservation stations is dispatched and this is controlled by the scheduler output sch_lsu_di?. As soon as an instruction is dispatched, it is 

---

3Multiple instructions can be simultaneously dispatched, executed and written back in a single clock cycle. However, for simplicity of proofs, multiple instruction issue or retirement is not allowed in a single clock cycle in this example. This is not an inherent limitation of the methodology (EX1.2 has multiple instruction issue and retirement).
its reservation station is freed. The dispatched instructions are *executed* in the ABS Unit and the LS Unit after a nondeterministic amount of time as determined by the scheduler outputs `sch_abs_ex`? (a ABS_su-wide vector) and `sch_lsu_ex`? respectively. During the execution, an *alu* instruction may raise an arithmetic exception, and *load* and *store* instructions in *user* mode may raise an exception if the memory address is outside the legal bounds maintained in *LOW_BOUND* and *HIGH_BOUND* special registers. After execution, at a time determined by the scheduler outputs `sch_abs_yb`? (a ABS_su-wide vector) and `sch_lsu_yb`?, the results are *written back* to their respective reorder buffer entries as well as forwarded to those instructions waiting for this result, if an exception is not raised by the executing instruction. If an exception is raised, then a flag is set in the reorder buffer entry to indicate that. In either case, a ready flag is set in the reorder buffer entry indicating that execution of the instruction is complete and it may be considered for retirement. A *store* instruction executes slightly differently when it does not raise an exception. It records the memory address and the value to be stored into the data memory in a store buffer entry rather than the reorder buffer entry. All the *store* instructions in the store buffer are maintained in their program order. When two store buffer entries refer to the same memory address, the “latest” one has a flag set. During the execution of a *load* instruction, the instruction first attempts an associative search in the store buffer using the memory address, and if multiple store buffer entries have the same address, the search returns the value of the “latest” entry. If the search does not find a matching entry, it then reads the value from the data memory.

The instruction at the tail of the reorder buffer is committed to the architecturally visible components at a time determined by `sch_retire_rb`?, if it is done executing, that is, if its ready flag is set. If it is a *store* instruction, then the entry at the tail of the store buffer (this entry corresponds to the instruction at the tail of the reorder buffer as both buffers keep the instructions in their program order) is marked committed. Committed store buffer entries are later written into the data memory at a time determined by `sch_sb_retire_mem`?. Also, if the RTT entry for the destination of the instruction being retired is pointing to the tail of the reorder buffer, then that RTT entry is reset to reflect the fact that the value of that register is in the appropriate register file. If the instruction at the tail of the reorder buffer has raised an exception or if it is a mispredicted branch or if it is a *prepare* instruction (the implementation asserts the *restart.proc*? signal), then the rest of the instructions in the reorder buffer are squashed, and the processor
is restarted by resetting all of its internal nonobservable state and setting the program counter to the appropriate instruction address.

5.2 Compositional Construction of the Abstraction Function

This section is organized in a manner paralleling Section 3.1.

5.2.1 Unfinished instructions and the program order

Every executing instruction in the processor has a reorder buffer entry associated with it, and all the executing instructions are stored in their program order in the reorder buffer. An instruction in the processor is identified by its reorder buffer index, that is, we refer to the instruction at reorder buffer index \( rbi \) as just instruction \( rbi \). In addition to these, the store buffer has certain committed store instructions yet to be written into the data memory, recorded in their program order. These store instructions are not associated with any reorder buffer entries and occur earlier in the program order than the instructions in the reorder buffer. We refer to these instructions by their store buffer index in the LS Unit.

5.2.2 Specifying the completion functions

An unfinished instruction \( rbi \) in the processor can be in one of the following seven phases: issued to ABS Unit (\( issued_{abs} \)), issued to LS Unit (\( issued_{lsu} \)), dispatched in ABS Unit (\( dispatched_{abs} \)), dispatched in LS unit (\( dispatched_{lsu} \)), executed in ABS Unit (\( executed_{abs} \)), executed in LS Unit (\( executed_{lsu} \)) or written back to the reorder buffer (\( writtenback \)). A given unfinished instruction is in exactly one of these phases at any given time. For each phase “ph”, we formulate a predicate “Instr\_ph?” describing an instruction in phase “ph”. For example, \( Instr\_{issu{\texttt{ed}}}_{abs}(q,rbi) \) is true exactly when \( rbi \) is in \( issued_{abs} \) phase in the implementation state \( q \). To facilitate this formulation, five auxiliary components are added to a reorder buffer entry. These components are \( rb\_{abs\_rs} \)

---

Footnote:

4 Brief explanation of the notation used throughout this chapter: \( q \) represents the current implementation state, \( s \) represents the current scheduler output and \( i \) represents the current processor input. If \( rf \) is an implementation component, \( rf(q) \) stands for its value in state \( q \), and will be referred to as just \( rf \) when the current implementation state is clear from the context. Similarly, when the current instruction \( rbi \) is clear from the context, \( rb\_{dest}(q)(rbi) \) will be referred to as just \( rb\_{dest} \). The "primed" notation is used to refer to the value of a component after an implementation transition, so \( rb\_{dest}' \) refers to \( rb\_{dest}(\text{t\_step}(q,s,i))(rbi) \). Some of the predicates/signals are referred to without their parameters when they are clear from the context, that is, \( Instr\_{issu{\texttt{ed}}}_{abs}? \) is TRUE implies \( Instr\_{issu{\texttt{ed}}}_{abs}? (q,rbi) \) is TRUE.
recording the reservation station index in the ABS Unit allocated to an instruction rbi, 
\texttt{rb.abs.eu} recording the execution unit index on which rbi is executing, \texttt{rb.lsu.rs} recording the reservation station index in the LS Unit allocated to rbi, \texttt{rb.lsu.eu} recording whether or not rbi is executing in the LS unit, and \texttt{rb.lsu.sb} recording the store buffer index allocated to rbi. The component \texttt{rb.abs.rs} is initialized to 0 and when rbi is issued to a reservation station in ABS Unit, \texttt{rb.abs.rs} records the index of that reservation station. When rbi is eventually dispatched onto an execution unit, \texttt{rb.abs.rs} is reset to 0. Other auxiliary components are modified in a similarly appropriate manner. \texttt{Instr.issued.abs?} predicate can then be defined as shown in \cite{1} and other predicates are similarly defined. These predicates refer to the nonobservable components only.

\begin{verbatim}
Instr.issued.abs?(q:state_I,rbi:rbindex): bool =
rb.abs.rs(q)(rbi) > 0
\end{verbatim}

We now identify how to complete an instruction in each of these phases and the conditions under which an instruction in each of these phases causes the subsequent instructions to be squashed. For each phase “ph”, we define a function “Action.ph” that specifies what ought to be the effect of completing an instruction in phase “ph” and a predicate “Squash.rest?.ph” that specifies the conditions under which an instruction in phase “ph” causes all the subsequent instructions to be squashed. The completion function and the squashing predicate are then defined as shown in \cite{2} following the template discussed in Chapter 3.

\begin{verbatim}
Complete_instr(q:state_I,rbi:rbindex,kill?:bool): state_I =
IF kill? THEN q
ELSIF Instr.writtenback?(q,rbi) THEN Action.writtenback(q,rbi)
ELSIF Instr.executed.lsu?(q,rbi) THEN Action.executed.lsu(q,rbi)
ELSIF Instr.executed.abs?(q,rbi) THEN Action.executed.abs(q,rbi)
ELSIF Instr.dispatched.lsu?(q,rbi) THEN Action.dispatched.lsu(q,rbi)
ELSIF Instr.dispatched.abs?(q,rbi) THEN Action.dispatched.abs(q,rbi)
ELSIF Instr.issued.lsu?(q,rbi) THEN Action.issued.lsu(q,rbi)
ELSIF Instr.issued.abs?(q,rbi) THEN Action.issued.abs(q,rbi)
ELSIF q ENDIF

Squash.rest?.instr(q:state_I,rbi:rbindex): bool =
IF Instr.writtenback?(q,rbi) THEN Squash.rest?.writtenback(q,rbi)
ELSIF Instr.executed.lsu?(q,rbi) THEN Squash.rest?.executed.lsu(q,rbi)
ELSIF Instr.executed.abs?(q,rbi) THEN Squash.rest?.executed.abs(q,rbi)
ELSIF Instr.dispatched.lsu?(q,rbi) THEN Squash.rest?.dispatched.lsu(q,rbi)
ELSIF Instr.dispatched.abs?(q,rbi) THEN Squash.rest?.dispatched.abs(q,rbi)
ELSIF Instr.issued.lsu?(q,rbi) THEN Squash.rest?.issued.lsu(q,rbi)
ELSIF Instr.issued.abs?(q,rbi) THEN Squash.rest?.issued.abs(q,rbi)
ELSIF FALSE ENDIF
\end{verbatim}
We now elaborate on specifying Action\_writtenback and Squash\_rest?\_writtenback as examples for illustration. In the EX2.2 implementation, an instruction squashes all the subsequent instructions if it can (eventually) raise an exception or if it is a rfeh instruction or if it (eventually) turns out to be a mispredicted branch. When an instruction rbi is in the writtenback phase, rb\_exception\_raised? (a component in the reorder buffer entry) will be set if and only if rbi has raised an exception and rb\_val contains the result computed by the instruction (the branch target address for a beq instruction or the return address for a rfeh instruction or the exception vector address when rbi raises an exception). rb\_rfeh? and rb\_beq? components record whether or not rbi is an rfeh or beq instruction respectively, and rb\_predicted bt component records the predicted branch target address for a branch instruction. Using this information, Squash\_rest?\_writtenback can be specified as shown in

```
Squash_rest?_writtenback(q:state_I, rbi: rbindex): bool =
    rb_exception_raised?(q(rbi)) OR rb_rfeh?(q(rbi)) OR
    (rb_beq?(q(rbi)) AND rb_val(q(rbi)) /= rb_predicted bt(q(rbi))
```

In this implementation, the six different observable components ought to be modified as follows when an instruction rbi completes its execution:

1. If rbi does not raise an exception and its destination register (if any) is a regular register, then that register in rf will contain the final result computed by rbi.

2. If rbi raises an exception, then the RESTART register in sp\_rf will contain the instruction memory address of rbi. Otherwise if the destination register of rbi is a special register, then that register in sp\_rf will contain the final result computed by rbi.

3. If rbi raises an exception, then the mode_flag will be changed to supervisory; otherwise if rbi is an rfeh instruction, it will be changed to user.

4. If rbi does not raise an exception and is a store instruction, then dmem will contain the value to be stored at the destination memory address.

5. If rbi causes the subsequent instructions to be squashed, then pc will be changed to either the exception vector address when rbi raises an exception or the return address of an rfeh instruction or the correct branch target for a beq instruction.
6. imem will not be changed.

We now turn to specifying Action_writtenback. rb_dest records the destination register of rbi, rb_reg_write?/rb_sp_reg_write? tell which register file is (possibly) modified, rb_pc records the instruction address. As pointed out earlier, for a store instruction, rb_lsu_sb records the index of the store buffer entry allocated, and lsu_sb_addr and lsu_sb_val components in that entry contain the memory address and the data value to be stored at that address. From the description above, Action_writtenback can be specified as shown in [4].

```
Action_writtenback(q, state_I, rbi, rbindex): state_I = q WITH [(rf) :=
    IF (NOT rb_exception_raised?(q)(rbi) AND rb_reg_write?(q)(rbi)) THEN
        assign(rf(q), rb_dest(q)(rbi), rb_val(q)(rbi))
    ELSE rf(q) ENDIF,
    (sp_rf) :=
        IF rb_exception_raised?(q)(rbi) THEN
            assign(sp_rf(q), RESTART, rb_pc(q)(rbi))
        ELSIF rb_sp_reg_write?(q)(rbi) THEN
            assign(sp_rf(q), rb_dest(q)(rbi), rb_val(q)(rbi))
        ELSE sp_rf(q) ENDIF,
    (mode_reg) :=
        IF rb_exception_raised?(q)(rbi) THEN
            Supervisory
        ELSIF rb_rfleh?(q)(rbi) THEN
            User
        ELSE mode_reg(q) ENDIF,
    (dmem) :=
        IF (NOT rb_exception_raised?(q)(rbi) AND rb_lsu_sb(q)(rbi) > 0) THEN
            write(dmem(q), value2mem_addr(lsu_sb_addr(q)(rb_lsu_sb(q)(rbi))),
            lsu_sb_val(q)(rb_lsu_sb(q)(rbi)))
        ELSE dmem(q) ENDIF,
    (pc) :=
        IF Squash_rest?_writtenback(q, rbi) THEN
            rb_val(q)(rbi)
        ELSE pc(q) ENDIF ]
```

The “Action” functions and “Squash_rest?” predicates for the other phases are specified in a similar manner and only some interesting details are discussed below. When an instruction rbi is in issued_abs phase, we need to determine its source operands to define Action_issued_abs. Consider the specification of the expected value for second source operand. In the reservation station allocated to rbi, the field abs_rs_ptr2 points to the instruction that produces the second source operand for rbi, and when it is 0, the source operand is ready and is available in abs_rs_val2. So when abs_rs_ptr2 is 0, the source operand is obtained from the component abs_rs_val2, otherwise it is obtained by reading
from the register file. This definition relies on the assumption that the instructions ahead of rbi are completed when we consider completing rbi, and this assumption is met while defining the abstraction function in Section 5.2.3. This specification is shown in and other source operands (the first source operand and the source operands when rbi is in issued phase) are specified similarly.

```plaintext
abs_src2_value(q:state_I,rbi:Issued_abs_instr(q)): value = 
    LET abs_rsi = rb_abs_rs(q)(rbi) IN 
    IF abs_rs_ptr2(q)(abs_rsi) = 0 THEN 
        abs_rs_val2(q)(abs_rsi) 
    ELSE select(rf(q),rb_dest(q)(abs_rs_ptr2(q)(abs_rsi))) ENDIF
```

We also need to specify the correct result computed by a load instruction the LS Unit.
This is specified as shown in and relies on the same assumption as before.

```plaintext
LSU_result_val(q:state_I,op:opcode,addr:value): value = 
    IF op = LOAD THEN 
        read(dmem(q),value2dmem_addr(addr)) 
    %% If not a LOAD instruction, this value doesn’t matter. 
    ELSE default_value ENDIF
```

We now consider how to complete the committed instructions in the store buffer. These instructions can be in only one phase (i.e., sb.commit), and they do not cause any of the subsequent instructions to be squashed. (Recall that a store instruction that raises an exception is not entered into the store buffer.) We define Action_sb_commit to complete such an instruction.

### 5.2.3 Constructing the abstraction function

The abstraction function is defined by flushing the pipeline, that is, by completing all the unfinished instructions in their program order. Defining the abstraction function in the above manner requires referring to the instructions in the reorder buffer in their program order. The reorder buffer index cannot be directly used for this purpose since the reorder buffer is implemented as a circular fifo queue. In order to refer to the instructions in their program order, we define a measure function rbi.measure_fn that associates a measure with every instruction in the reorder buffer such that the instruction at the tail has measure one and successive instructions have a measure one greater than the previous

---

5An alternate approach to specifying the source operands is possible such as reading from the observable whether or not the source operand is ready (used in and also for the load instructions here), and this approach necessitates proving different invariants/obligations on the feedback logic.
instruction. So instructions with lower measure occur earlier in the program order. The function \texttt{measure\_fn\_rbi} returns the reorder buffer index of the instruction with a given measure and these definitions are shown in \textbf{7}. We refer to an instruction either by its reorder buffer index \texttt{rbi} or by its measure \texttt{rbi\_ms}. To define the abstraction function, we first define a recursive function \texttt{Complete\_Squash\_rest?\_till} that completes the instructions and computes the disjunction of the squashing predicates from the tail of the reorder buffer till a given unfinished instruction, as shown in \textbf{7}. Observe that the \texttt{kill?} argument of \texttt{Complete\_instr} is specified to be the disjunction of the \texttt{Squash\_rest?\_instr} predicates for the instructions occurring earlier. \texttt{Complete\_committed\_in\_sb\_till} is a similar recursive function that completes the committed \texttt{store} instructions in the store buffer from the tail till a given store buffer index. We can now define the abstraction function by completing all the committed \texttt{store} instructions in the store buffer (recall that these occur earlier in the program order than any instruction in the reorder buffer) and then completing all the unfinished instructions in the reorder buffer. \texttt{rb\_count} maintains the number of instructions in the reorder buffer and hence corresponds to the measure of the latest instruction. Similarly, \texttt{lsu\_sb\_commit\_count} (an auxiliary component in this example) maintains the number of committed instructions in the store buffer and hence corresponds to the measure of the latest committed \texttt{store} instruction. We first define \texttt{Complete\_till} and \texttt{Squash\_rest?\_till} as shown in \textbf{7}, and then in defining the abstraction function \texttt{ABS}, we instantiate the \texttt{Complete\_till} definition with the measure of the latest instruction in the reorder buffer, that is, \texttt{rb\_count}.

5.2.4 Defining the synchronization function

The synchronization function returns zero under the following conditions:

- In a cycle in which the processor is being restarted (no new instructions are fetched in such a cycle).
- If any of the instructions in the reorder buffer causes the rest of the instructions to be squashed (the new instruction is squashed too).
- When no new instruction is issued either because \texttt{no\_op?} is asserted or there is no free reservation station/reorder buffer location to hold the new instruction.
Otherwise the synchronization function returns one. Its definition is shown

```plaintext
rbi_measure_fn(q:state_I,rbi:rbindex): rbindex =  7
    IF rbi >= rb_tail(q) THEN rbi - rb_tail(q) + 1
    ELSE rbi - rb_tail(q) + 1 + RB ENDIF
measure_fn_rbi(q:state_I,ms:rbindex): rbindex =
    LET ct = rb_tail(q) + ms - 1 IN
    IF ct > RB THEN ct - RB
    ELSE ct ENDIF
Complete_Squash_rest?_till(q:state_I,rbi_ms:rbindex_p): RECURSIVE [state_I,bool] =
    IF rbi_ms = 0 THEN (q,FALSE)
    ELSE LET t = Complete_Squash_rest?_till(q,rbi_ms-1),
        x = proj_1(t), y = proj_2(t) IN
        (Complete_instr(x,measure_fn_rbi(q,rbi_ms),y),
        Squash_rest?_instr(x,measure_fn_rbi(q,rbi_ms)) OR y) % Second component.
    ENDIF
MEASURE rbi_ms
Complete_till(q:state_I,rbi_ms:rbindex_p): state_I =
    proj_1(Complete_Squash_rest?_till(
        Complete_committed_in_sb_till(q,lsu_sb_commit_count(q)),rbi_ms))
Squash_rest?_till(q:state_I,rbi_ms:rbindex_p): bool =
    proj_2(Complete_Squash_rest?_till(
        Complete_committed_in_sb_till(q,lsu_sb_commit_count(q)),rbi_ms))
ABS(q: state_I): state_A =
    projection(Complete_till(q,rb_count(q)))
```

```plaintext
sync_fun(q:state_I,s:schedule,i:inputs_type): nat =  8
    IF restart_proc?(q,s) THEN 0
    ELSIF Squash_rest?_till(q,rb_count(q)) THEN 0
    ELSIF not_ifu_instr_issued?(q,s,i) THEN 0
    ELSE 1 ENDIF
```

5.3 Decomposing the Proof

This section discusses the formulation of the different verification conditions, the invariant properties needed about the implementation machine, and proof strategies used in discharging the different proof obligations.

---

6Many of the signatures like that of I_step, sync_fun, etc. shown in Chapter 2 will take the scheduler output s as yet another argument.
5.3.1 Formulating the different verification conditions

This section explains the different verification conditions and some other lemmas needed in the proof of the commutativity obligation. As explained in Chapter 3, the different verification conditions are generated by attempting to relate the values of the observable components, the squashing predicate and the source operands of the instructions in the present and the next implementation states. They are then decomposed into different cases based on how an instruction makes a transition from its present phase to the next phase.

This section is organized into different subsections, discussing the verification conditions for the following cases:

1. We discuss the verification conditions for the case when the processor is being restarted. (This is an easy case in the proof of the commutativity obligation.)

2. We then assume that the processor is not being restarted, and discuss:

   (a) A parameterized verification condition due to the unfinished instructions in the reorder buffer for the observable components \( \text{rf, sp.rf, mode.reg} \) and \( \text{dmem} \), and the \( \text{Squash.rest? till} \) predicate, and how the proof of this verification condition is decomposed, and an additional verification condition for \( \text{mode.reg} \) due to these unfinished instructions.

   (b) The verification conditions regarding the correctness of the source operands and due to the feedback logic.

   (c) The verification conditions for the observable \( \text{pc and imem} \) due to the unfinished instructions in the reorder buffer.

   (d) The verification condition due to the new instruction being issued for all the observable components and regarding the correctness of its source operands.

   (e) A parameterized verification condition due to the committed instructions in the store buffer.

5.3.1.1 Verification conditions for case 1

We first consider an easy case in the proof of the commutativity obligation, that is, when the processor is being restarted in the current cycle (\( \text{restart.proc is TRUE} \)). The verification conditions generated in this case are shown in 9.
• When the processor is being restarted, the instruction at the tail of the reorder buffer is being committed to the observable components, and there are no committed store instructions in the store buffer. So the contents of the observable components in \texttt{I\_step}(q,s,i) should be the same as after completing the instruction at the tail of the reorder buffer. This verification condition for \texttt{rf} is shown as \texttt{Restarting\_rf\_rb\_tail\_VC} in \ref{9}. The verification conditions for \texttt{sp\_rf}, \texttt{mode\_reg}, \texttt{dmem} and \texttt{pc} are similar, and one for \texttt{imem} states that it is unchanged.

• When the processor is being restarted, the instruction at the tail of the reorder buffer is causing the rest of the instructions to be squashed. So \texttt{Squash\_rest\_till}(q,1) ought to be \texttt{TRUE} (\texttt{Restarting\_Squash\_rest\_till\_rb\_tail\_VC} shown in \ref{9}).

<table>
<thead>
<tr>
<th>Restarting_rf_rb_tail_VC: CLAIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{restart_proc}(q,s) IMPLIES \texttt{rf(Complete_till(q,1)) = rf(I_step(q,s,i))}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Restarting_Squash_rest_till_rb_tail_VC: CLAIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{restart_proc}(q,s) IMPLIES \texttt{Squash_rest_till}(q,1)</td>
</tr>
</tbody>
</table>

\subsection{5.3.1.2 Verification conditions for case 2(a)}

We postulate a parameterized verification condition that applies to all the pending instructions in the reorder buffer that relates the effect of completing the instructions in the reorder buffer in the present and the next implementation states. Consider an arbitrary unfinished instruction \texttt{rbi}. Consider the effect on the observable components \texttt{rf}, \texttt{sp\_rf}, \texttt{mode\_reg} and \texttt{dmem} of completing all the instructions till \texttt{rbi} in the present and the next implementation states. Though the processor executes the instructions in an out-of-order manner, it commits the instructions to these observable components only in their program order. This suggests that the effect on these observable components of completing all the instructions till \texttt{rbi} is the same in states \texttt{q} and \texttt{I\_step}(q,s,i). Similarly the truth or the falsity of whether an instruction causes the rest of the instructions to be squashed remains the same in the present and the next implementation states. So the \texttt{Squash\_rest\_till} predicate ought to have the same truth value in states \texttt{q} and \texttt{I\_step}(q,s,i). We introduce a definition \texttt{rf\_sp\_rf\_mode\_reg\_dmem\_Squash\_rest\_till\_remains\_same} that captures these observations as shown in \ref{10}. In this definition, \texttt{rbi\_measure\_adjust} is a function that adjusts the measure so that it refers to the same instruction after an implementation transition.
This adjustment is necessary because the measure of an instruction depends on the tail of the reorder buffer, and the tail can change during an implementation transition. The verification condition `rf_sp_rf_mode_reg_dmem_Squash_rest?_till_remains_same_VC` is also shown in [10]. It holds only for valid instructions in the reorder buffer, that is, for those instructions within the reorder buffer bounds, and `valid_rb_entry?` predicate tests for this condition.

```plaintext
rbi_measure_adjust(q:state_I,s:schedule,rbi_ms:rbindext): rbindext_p =
  IF NOT sch_retire_rb?(s) THEN rbi_ms
  ELSE rbi_ms - 1 ENDIF

rf_sp_rf_mode_reg_dmem_Squash_rest?_till_remains_same(q:Reach_state_I,
  s:schedule,i:inputs_type,rbi_ms:rbindext): bool =
  rf(Complete_till(q,rbi_ms)) = rf(Complete_till(I_step(q,s,i),
    rbi_measure_adjust(q,s,rbi_ms)))
  AND
  sp_rf(Complete_till(q,rbi_ms)) = sp_rf(Complete_till(I_step(q,s,i),
    rbi_measure_adjust(q,s,rbi_ms)))
  AND
  mode_reg(Complete_till(q,rbi_ms)) = mode_reg(Complete_till(I_step(q,s,i),
    rbi_measure_adjust(q,s,rbi_ms)))
  AND
  dmem(Complete_till(q,rbi_ms)) = dmem(Complete_till(I_step(q,s,i),
    rbi_measure_adjust(q,s,rbi_ms)))
  AND
  Squash_rest?_till(q,rbi_ms) = Squash_rest?_till(I_step(q,s,i),
    rbi_measure_adjust(q,s,rbi_ms))

rf_sp_rf_mode_reg_dmem_Squash_rest?_till_remains_same_VC: CLAIM
  FORALL (rbi_ms:rbindext):
    LET rbi = measure_fn_rbi(q,rbi_ms) IN
    (valid_rb_entry?(q,rbi) AND NOT restart_proc?(q,s))
    IMPLIES
    rf_sp_rf_mode_reg_dmem_Squash_rest?_till_remains_same(q,s,i,rbi_ms)
```

We prove `rf_sp_rf_mode_reg_dmem_Squash_rest?_till_remains_same_VC` by an induction on `rbi_ms`. This proof is decomposed based on how an instruction makes a transition from its present phase to its next phase. Figure 5.2 shows the phase transitions for an instruction `rbi` in the reorder buffer (when the processor is not being restarted) where the predicates labeling the arcs define the conditions under which those transitions take place. The figure also shows the three transitions (Issue...) for a new instruction entering the processor pipeline. `Dispatch_abs?(q,s,rbi)` predicate, for example, defines the conditions under which the instruction `rbi` goes from `issued_abs` phase to `dispatched_abs` phase. In this implementation, this predicate is TRUE when there is an execution unit for which `sch.abs.di?` output from the scheduler is TRUE.
Figure 5.2. Various phases an instruction can be in and transitions between them (when the processor is not being restarted)

and the sch.abs.di output is equal to the reservation station index assigned to rbi. (The scheduler will assert these signals only if the execution unit is free and the source operands for rbi are ready.) Similarly other “transition” predicates are defined.

Having defined these predicates, we need to prove that they indeed cause the instructions to take the transitions shown. As an example, consider a valid instruction rbi in executed.abs (E.abs in Figure 5.2) phase. If Writeback.abs predicate is TRUE, then after an implementation transition, rbi will be in writtenback phase and remains valid. This is shown as lemma abs.executed_to_writtenback in [11]. If Writeback.abs predicate is FALSE, then rbi remains in the executed.abs phase and remains valid after an implementation transition. This is shown as lemma abs.executed_remains in [11]. We also prove certain other relevant properties needed later in the proofs. In both cases, the adjusted measure is nonzero, and in the second case, the execution unit assigned to rbi does not change while rbi remains in the executed.abs phase. The lemmas for all the other transitions are similar except for the case when an instruction in writtenback phase is being retired. In this case, the instruction will not be valid after an implementation transition and the adjusted measure is zero. All together, there are 17 lemmas for the 17 transitions shown in Figure 5.2.

Now in the proof of rf. sp.rf.mode_reg.dmem.squash.rest?.till.remains same.VC, we do a case analysis on the possible phases rbi is in, and whether or not, it makes a transition to its next phase. As an example, assume that rbi is in issued.abs phase (I.abs in Figure 5.2). Now there are two cases depending on whether Dispatch.abs? is TRUE or FALSE. The proof obligation for the second case is shown in [12]. There are similar
proof obligations for the other phases rbi can be in. Overall, the proof decomposes into 14 obligations. \[12\] also shows the induction hypothesis rb_induction_hypothesis.

```plaintext
abs_executed_to_writtenback: LEMMA
FORALL (rbi_ms: rbindex):
LET rbi = measure_fn_rbi(q, rbi_ms) IN
(valid_rub_entry?(q, rbi) AND NOT restart_proc?(q, s) AND
Instr_executed_abs?(q, rbi) AND Writeback_abs?(q, s, rbi))
IMPLIES
(Instr_writtenback?(I_step(q, s, i), rbi) AND
valid_rub_entry?(I_step(q, s, i), rbi) AND
rbi_measure_adjust(q, s, rbi_ms) /= 0)

abs_executed_remains: LEMMA
FORALL (rbi_ms: rbindex):
LET rbi = measure_fn_rbi(q, rbi_ms) IN
(valid_rub_entry?(q, rbi) AND NOT restart_proc?(q, s) AND
Instr_executed_abs?(q, rbi) AND NOT Writeback_abs?(q, s, rbi))
IMPLIES
(Instr_executed_abs?(I_step(q, s, i), rbi) AND
valid_rub_entry?(I_step(q, s, i), rbi) AND
rbi_measure_adjust(q, s, rbi_ms) /= 0 AND
rb_abs_eu(I_step(q, s, i))(rbi) = rb_abs_eu(q)(rbi))

abs_issued_remains_induction: CLAIM
FORALL (rbi_ms: rbindex):
LET rbi = measure_fn_rbi(q, rbi_ms) IN
(valid_rub_entry?(q, rbi) AND NOT restart_proc?(q, s)
AND Instr_issued_abs?(q, rbi) AND NOT Dispatch_abs?(q, s, rbi)
AND rb_induction_hypothesis(q, s, i, rbi_ms))
IMPLIES
rf_sp_rf_mode_reg_dsem_Squash_rest?_till_remains_same(q, s, i, rbi_ms)

rb_induction_hypothesis(q: Reach_state I, s: schedule, i: inputs_type,
                      rbi_ms: rbindex): bool =
(1 <= rbi_ms-1 AND rbi_ms-1 <= RB)
IMPLIES
((valid_rub_entry?(q, measure_fn_rbi(q, rbi_ms-1)) AND
  NOT restart_proc?(q, s))
IMPLIES
rf_sp_rf_mode_reg_dsem_Squash_rest?_till_remains_same(q, s, i, rbi_ms-1))
```

We formulate an additional verification condition for mode_reg due to the unfinished instructions in the reorder buffer. An executing instruction modifies mode_reg only when it (eventually) raises an exception or if it is a rfef instruction. In both cases, it causes the rest of the instructions to be squashed. So when Squash_rest?_till predicate is FALSE for a given instruction rbi, completing the instructions till rbi ought to leave mode_reg unchanged. This observation is formulated as verification condition mode_reg_unchanged_VC, shown in \[13\]. load and store instructions enforce the bound
check for the memory addresses only in user mode. So during execution, these instructions read the current value of mode_reg while they should have read the value only after all the instructions ahead of it have completed execution or are guaranteed not to modify mode_reg. Now a load or a store instruction matters only if it is not squashed by an earlier instruction, and in such a case, from mode_reg_unchanged_VC, the two values of mode_reg above are equivalent. That is, load and store instructions use mode_reg as a source operand, and mode_reg_unchanged_VC asserts the correctness of this source operand.

```
mode_reg_unchanged_VC: CLAIM
FORALL(rbi_ms:rbindex_p):
  NOT Squash_rest?_till(q,rbi_ms) IMPLIES
  mode_reg(Complete_till(q,rbi_ms)) = mode_reg(q)
```

### 5.3.1.3 Verification conditions for case 2(b)
We first describe how to generate the verification conditions expressing the correctness of the feedback logic for both the register file and the data memory as described in Chapter 3.

Consider an instruction rbi in issued_abs phase which remains in that phase after an implementation transition (i.e., Dispatch_abs? is FALSE). As explained earlier, field abs_rs_ptr2 in the reservation station allocated to rbi points to the instruction that produces the second source operand for rbi. Assume that abs_rs_ptr2 is nonzero and it becomes zero in the current implementation transition. Then in the current implementation cycle, the instruction pointed to by abs_rs_ptr2 forwards its result to rbi. So the value in abs_rs_val2 after an implementation transition ought to be equal to the expected value as given by the specification abs_src2.value (see [5]). This observation is formulated as verification condition abs_src2.value_feedback_VC in [14]. There are similar verification conditions for the first source operand and for the four source operands of an instruction in issued_lsu phase. (The load and store instructions read two bound registers too.)

The verification condition due to the feedback logic for data memory is similar. Consider a load instruction rbi that is in dispatched_lsu phase and making a transition to executed_lsu phase in the current cycle. rbi obtains its result value from the store buffer if there is an entry with the matching address (using associative search), otherwise it reads the value from the data memory. So the value in lsu.eu_result (the implementation component where the result is deposited) after the current implementation transition
ought to be equal to the expected value given by the specification for the result of a load instruction. (See 6.) This observation is formulated as verification condition load_value_feedback_VC in 14.

\[
\text{abs_src2_value_feedback_VC: CLAIM} \quad \begin{align*}
\text{FORALL (rbi, ms: rbindex):} \\
\text{LET } rbi = \text{measure_fn_rbi}(q, rbi, ms) \text{ IN} \\
\text{(valid_rb_entry?(q, rbi) AND NOT restart_proc?(q, s) \quad \text{AND Instr issued.abs?(q, rbi) AND NOT Dispatch_abs?(q, s, rbi) \quad AND NOT Squash_rest?_till(q, rbi, ms-1) \quad AND abs_rs_ptr2(q)(rb_abs_rs(q)(rbi)) \neq 0 \quad \text{AND abs_rs_ptr2(I_step(q, s, i))(rb_abs_rs(q)(rbi)) = 0) \quad I}\text{MPLIES} \\
\text{next_abs_rs_val2(q, s, i)(rb_abs_rs(q)(rbi)) =} \\
\text{abs_src2_value(Complete_till(q, rbi, ms-1), rbi)}
\end{align*}
\]

\[
\text{load_value_feedback_VC: CLAIM} \quad \begin{align*}
\text{FORALL (rbi, ms: rbindex):} \\
\text{LET } rbi = \text{measure_fn_rbi}(q, rbi, ms), \\
\text{address = d_add(lsu.eu_val(q), lsu.eu_immed(q)) \text{ IN} \\
\text{(valid_rb_entry?(q, rbi) AND NOT restart_proc?(q, s) \quad \text{AND Instr dispatched.lsu?(q, rbi) AND Execute_lsu?(q, s, rbi) \quad AND NOT Squash_rest?_till(q, rbi, ms-1) \quad AND NOT LSU\_exception?(mode_reg(q), address, lsu.eu.ib_val(q), \quad lsu.eu_ub_val(q))) \quad I}\text{MPLIES} \\
\text{next_lsu.eu_result(q, s) =} \\
\text{LSU\_result_val(Complete_till(q, rbi, ms-1), lsu.eu.op(q), address)}
\end{align*}
\]

We now consider the verification conditions regarding the correctness of the source operands of the instructions. Consider again an instruction rbi in issued_abs phase. As pointed out before, when an instruction is in issued_abs phase, the expected value of the second source operand is given by abs_src2_value. Assume that rbi remains in the issued_abs phase after an implementation transition. Then the second source operand value as given by the specification abs_src2_value ought to remain same in the present and the next implementation states (after the instructions ahead of rbi are completed). This verification condition is shown as abs_src2_value_same_VC in 15.

\[
\text{abs_src2_value_same_VC: CLAIM} \quad \begin{align*}
\text{FORALL (rbi, ms: rbindex):} \\
\text{LET } rbi = \text{measure_fn_rbi}(q, rbi, ms) \text{ IN} \\
\text{(valid_rb_entry?(q, rbi) AND NOT restart_proc?(q, s) \quad \text{AND Instr issued.abs?(q, rbi) AND NOT Dispatch_abs?(q, s, rbi) \quad AND NOT Squash_rest?_till(q, rbi, ms-1) \quad AND rb_induction_hypothesis(q, s, i, rbi, ms) \quad I}\text{MPLIES} \\
\text{abs_src2_value(Complete_till(q, rbi, ms-1), rbi) =} \\
\text{abs_src2_value(Complete_till(I\_step(q, s, i), \quad rbi_measure_adjust(q, s, rbi, ms)-1), rbi)}
\end{align*}
\]
5.3.1.4 Verification conditions for case 2(c)

The verification conditions for the program counter are slightly involved as explained in Chapter 3. Due to its being modified at two places in the pipeline, it is not possible to relate its value in the present and the next implementation states by considering the effect of completing the instructions \textit{one at a time in their program order} as was done for, say, \texttt{rf}.

An instruction executing in the pipeline will not modify \texttt{pc} (when completed) if it does not cause the rest of the instructions to be squashed. This observation leads to the verification condition \texttt{pc.unchanged.VC}, which is similar to \texttt{mode.reg.unchanged.VC}. When one of the instructions under consideration causes the rest of the instructions to be squashed, we can relate the \texttt{pc} values in states \texttt{q} and \texttt{I.step(q,s,i)}. Such an instruction, when completed, updates \texttt{pc} and the subsequent instructions will not be of interest, and it ought to update \texttt{pc} in the same way in both \texttt{q} and \texttt{I.step(q,s,i)}. This observation leads to the verification condition \texttt{pc.remains.same.VC1} shown in [16]. Even when none of the instructions under consideration causes the rest of the instructions to be squashed, we can still relate the \texttt{pc} values in states \texttt{q} and \texttt{I.step(q,s,i)} if no new instruction is being issued into the pipeline. This observation leads to the verification condition \texttt{pc.remains.same.VC2} shown in [16].

<table>
<thead>
<tr>
<th>pc.unchanged.VC: CLAIM</th>
<th>\texttt{FORALL(rbi_ms:rbindex_p)}:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{NOT Squash_rest_till(q,rbi_ms)} IMPLIES</td>
<td></td>
</tr>
<tr>
<td>\texttt{pc(Complete_till(q,rbi_ms)) = pc(q)}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pc.remains.same.VC1: CLAIM</th>
<th>\texttt{FORALL(rbi_ms:rbindex)}:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{LET rbi = measure_fn_rbi(q,rbi_ms) IN}</td>
<td></td>
</tr>
<tr>
<td>(valid_rb_entry?(q,rbi) AND NOT restart_proc?(q,s) AND</td>
<td></td>
</tr>
<tr>
<td>Squash_rest_till(q,rbi_ms)) IMPLIES</td>
<td></td>
</tr>
<tr>
<td>\texttt{pc(Complete_till(q,rbi_ms)) = pc(Complete_till(I_step(q,s,i)),}</td>
<td></td>
</tr>
<tr>
<td>rbi_measure_adjust(q,s,rbi_ms)))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pc.remains.same.VC2: CLAIM</th>
<th>\texttt{FORALL(rbi_ms:rbindex)}:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{LET rbi = measure_fn_rbi(q,rbi_ms) IN}</td>
<td></td>
</tr>
<tr>
<td>(valid_rb_entry?(q,rbi) AND NOT restart_proc?(q,s) AND</td>
<td></td>
</tr>
<tr>
<td>\texttt{NOT Squash_rest_till(q,rbi_ms) AND NOT ifu_instr_issued?(q,s,i)}</td>
<td></td>
</tr>
<tr>
<td>IMPLIES</td>
<td></td>
</tr>
<tr>
<td>\texttt{pc(Complete_till(q,rbi_ms)) = pc(Complete_till(I_step(q,s,i)),}</td>
<td></td>
</tr>
<tr>
<td>rbi_measure_adjust(q,s,rbi_ms)))</td>
<td></td>
</tr>
</tbody>
</table>

\texttt{pc.remains.same.VC1} is proved by an induction on \texttt{rbi\_ms}, and is decomposed into 14 obligations similar to \texttt{rf.sp.rf.mode.reg.dmem.Squash\_rest\_till\_remains\_same.VC}. 
Finally, the instruction memory is not modified at all by any of the executing instructions. So it ought to remain unchanged when the unfinished instructions in the reorder buffer are completed. This verification condition \texttt{imem\_unchanged\_VC} is shown in \[17\].

\begin{verbatim}
imem\_unchanged\_VC: CLAIM
  FORALL (rbi\_ms:rbindex\_p): imem(Complete\_till(q,rbi\_ms)) = imem(q)
\end{verbatim}

\subsection{Verification conditions for case 2(d)}

When a new instruction issued into the pipeline is completed, it ought to have the same effect on the observable components as given by the ISA level specification. This observation leads to the verification condition \texttt{new\_instr\_issued\_rf\_VC} for register file as shown in \[18\]. The verification condition for \texttt{sp\_rf}, \texttt{mode\_reg}, \texttt{dmem} and \texttt{pc} are similar. The verification condition for \texttt{imem} is shown in \[18\]. \texttt{new\_instr\_issued\_rf\_VC} (and the verification conditions for the other observable components except \texttt{imem}) is decomposed into three obligations corresponding to the three transitions for a new instruction entering the pipeline shown in Figure 5.2.

\begin{verbatim}
new\_instr\_issued\_rf\_VC: CLAIM
  (NOT restart\_proc?(q,s) AND NOT Squash\_rest?\_till(q,rb\_count(q))
   AND ifu\_instr\_issued?(q,s,i)) IMPLIES
  rf(A\_step(projection(Complete\_till(q,rb\_count(q))))) =
  rf(projection(Complete\_till(1\_step(q,s,i),rb\_count(1\_step(q,s,i))))))
\end{verbatim}

\begin{verbatim}
new\_instr\_issued\_imem\_VC: CLAIM
  imem(A\_step(projection(Complete\_till(q,rb\_count(q))))) = imem(q)
\end{verbatim}

We formulate verification conditions on the source operands being read when a new instruction is issued. (These are similar to the verification conditions due to the feedback logic and correspond to the “no dependency” case.) Consider a new instruction being issued to the ABS Unit. If any of the instructions executing in the reorder buffer is going to write its source register (say, the second one) and has not yet finished execution, then the new instruction records the index of that instruction in \texttt{abs\_rs\_ptr2} and waits for the result to be forwarded, otherwise it reads the source value from the register file and makes \texttt{abs\_rs\_ptr2} zero. So for a new instruction being issued, if \texttt{abs\_rs\_ptr2} is zero, then \texttt{abs\_rs\_val} ought to be equal to the expected value of the second source operand as given by the specification machine \texttt{spec\_src2\_value}. This verification condition is shown as \texttt{new\_instr\_issued\_abs\_src2\_value\_read\_VC} in \[19\]. There are similar verification conditions for the first source operand and for the four source operands for an instruction being issued to LS Unit.
We then formulate verification conditions regarding the correctness of the source operands for the new instruction (whether or not the source operands are read in the current transition). Consider the second source operand value of the new instruction as given by spec.src2.value in state q and its value when that instruction is in issued.abs phase as given by abs.src2.value in state I_step(q,s,i). These two values ought to be equal and this observation is formulated as lemma new_instrissued.abs.src2.value_same_VC.

```plaintext
new_instrissued.abs.src2.value_read_VC: CLAIM
(NOT restart_proc?(q,s) AND Issued_abs?(q,s,i)
AND NOT Squash_rest?_till(q,rb_count(q))
AND abs.rs_ptr2(I_step(q,s,i))(sch.abs_slot(s)) = 0) IMPLIES
next_abs.rs_val2(q,s,i)(sch.abs_slot(s)) =
spec.src2.value(projection(Complete_till(q,rb_count(q))))
```

5.3.1.6 Verification conditions for case 2(e)

We postulate a verification condition store_buffer.dmem_remains_same_VC (similar to rf_sp.rf_mode_reg.dmem.Squash_rest?_till_remains_same_VC) that applies to the committed store instructions in the store buffer and the observable component dmem. This verification condition is shown in 20, and is proved by an induction on lsu.sbi.ms. A committed store instruction either remains in the the store buffer or gets written to data memory during an implementation transition, and this leads to two cases in the proof of store_buffer.dmem_remains_same_VC. The other observable components are unaffected and the verification condition for one of them (say rf) is also shown in 20.

```plaintext
store_buffer.dmem_remains_same_VC: CLAIM
FORALL (lsu.sbi.ms:LSU_sbiindex):
  LET lsu.sbi = measure_fn_lsu.sbi(q,lsu.sbi.ms) IN
  (valid_lsu.sb_entry?(q,lsu.sbi) AND Instr_sb_commit?(q,lsu.sbi)
   AND NOT restart_proc?(q,s))
  IMPLIES
dmem(Complete_committed_in_sb_till(q,lsu.sbi.ms)) =
dmem(Complete_committed_in_sb_till(I_step(q,s,i),
    lsu.sbi_measure_adjust(q,s,lsu.sbi.ms)))
```

```plaintext
store_buffer_rf_unchanged_VC: CLAIM
FORALL (lsu.sbi.ms:LSU_sbiindex_p):
  rf(Complete_committed_in_sb_till(q,lsu.sbi.ms)) = rf(q)
```
Table 5.1 shows the summary of all the verification conditions generated in the overall proof decomposition discussed so far. In addition, there are 17 lemmas corresponding to the 17 instruction phase transitions shown in Figure 5.2.

5.3.2 Invariants and other properties needed

This section provides a classification and a brief description of the different invariant properties about the implementation machine needed in our proof. Some other properties proved about the implementation machine that follow from these invariants or otherwise are also discussed.

5.3.2.1 Exclusiveness and exhaustiveness of instruction phases

Earlier we identified the different possible phases for an instruction in the reorder buffer. We formulate two properties regarding these instruction phases: 1) every instruction in the reorder buffer is in one of these phases at all times (Instruction_exhaustive invariant) and 2) a given instruction is never simultaneously in two of the phases at any time (Instruction_exclusive invariant). These properties ensure that, in the definitions of Complete_instr and Squash_rest?_instr, one and only one case always applies.

5.3.2.2 Instruction phase properties

We introduced auxiliary components to keep track of the indices of the different resources allocated to an instruction. We formulate certain properties on these auxiliary components, that they indeed track of what they ought to. The auxiliary component \( rb_{abs,rs} \) keeps track of the reservation station in the ABS Unit allocated to an instruction. Also, in a reservation station, the component \( abs_{rs}.valid \) indicates whether or not it is allocated to some instruction (the reservation station is said to be “valid”), and the component \( abs_{rs}.dest \) points to the reorder buffer entry of the instruction to which the reservation station is assigned. When an instruction \( rbi \) is in \( issued.abs \) phase (i.e., \( rb_{abs,rs} \) is greater than zero), the reservation station allocated to it ought to be valid and \( abs_{rs}.dest \) component in that reservation station ought to point to \( rbi \). Conversely, when a reservation station \( abs_{rsi} \) in the ABS Unit is valid, the reorder buffer entry it points to is valid and the \( rb_{abs,rs} \) component in that reorder buffer entry ought to point to \( abs_{rsi} \). These two properties are shown in [21]. There are similar properties corresponding to the other four auxiliary variables too.
Table 5.1. Verification conditions generated for EX2.2

<table>
<thead>
<tr>
<th>Verification condition regarding</th>
<th>Number of verification conditions and comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>rf</td>
<td>1 for the case when the processor is being restarted, 1 due to the unfinished instructions in the reorder buffer decomposed into 14 cases, 1 due to the new instruction being issued decomposed into 3 cases, 1 due to the committed instructions in the store buffer</td>
</tr>
<tr>
<td>sp_rf</td>
<td>Similar to above</td>
</tr>
<tr>
<td>mode_reg</td>
<td>Similar to above. And an additional one due to the unfinished instructions in reorder buffer</td>
</tr>
<tr>
<td>dmem</td>
<td>1 for the case when the processor is being restarted, 1 due to the unfinished instructions in the reorder buffer decomposed into 14 cases, 1 due to the new instruction being issued decomposed into 3 cases, 1 due to the committed instructions in the store buffer decomposed into 2 cases</td>
</tr>
<tr>
<td>pc</td>
<td>1 for the case when the processor is being restarted, 3 due to the unfinished instructions in the reorder buffer and one of them is decomposed into 14 cases, 1 due to the new instruction being issued decomposed into 3 cases, 1 due to the committed instructions in the store buffer</td>
</tr>
<tr>
<td>imem</td>
<td>1 for the case when the processor is being restarted, 1 due to the unfinished instructions in the reorder buffer, 1 due to the new instruction being issued, 1 due to the committed instructions in the store buffer</td>
</tr>
<tr>
<td>Squash_rest?_till predicate</td>
<td>1 for the case when the processor is being restarted, 1 due to the unfinished instructions in the reorder buffer decomposed into 14 cases</td>
</tr>
<tr>
<td>Correctness of the feedback logic and the source operands</td>
<td>1 on the correctness of the feedback logic for data memory, 6 on the correctness of the feedback logic for register file for the 6 operands, 6 on the correctness of the source operands for an instruction in <code>issued_abs/issued_lsu</code> phase, 6 on the correctness of the value read by the new instruction when no dependencies exist, 6 on the correctness of the source operands for the new instruction</td>
</tr>
</tbody>
</table>
5.3.2.3 Program order property in the LS Unit

The LS Unit executes the load and store instructions in their program order. So it maintains the instructions in the store buffer, the one instruction in the execution unit, and the instructions in the reservation stations in the LS Unit in their program order, from the earliest to the latest. This property of the LS Unit is needed later in the proofs, and it is formulated as follows for the instructions in the reservation stations in the LS Unit: The field `lsu.rs.dest` in a reservation station in LS Unit points to the reorder buffer entry to which the reservation station is assigned. Let `lsu.rs1` and `lsu.rs2` be two valid reservation station entries in the LS Unit. Let `rbi1` and `rbi2` be the instructions (reorder buffer indices) pointed to by these entries. `lsu.rs1.measure_fn` is a measure function defined on the queue of reservation stations similar to `rbi.measure_fn`. If `lsu.rs1` has a lower/higher measure than `lsu.rs2` according to `lsu.rs1.measure_fn`, then `rbi1` has a lower/higher measure than `rbi2` according to `rbi.measure_fn`. This property `LSU.rs.instr.in.order` is shown in Figure 22. There is a similar property stating that the instructions in the store buffer are in program order. Also,

- Any instruction in the store buffer is earlier in the program order than any instruction in the reservation station in the LS Unit.

- Any instruction in the store buffer is earlier in the program order than the instruction in the execution unit in the LS Unit.

- The instruction in the execution unit in the LS Unit is earlier in the program order than any instruction in the reservation station in the LS Unit.
5.3.2.4 Properties about the committed instructions in the store buffer

Whenever a store instruction at the tail of the reorder buffer is retired, the corresponding entry in the store buffer is marked committed (and written to data memory at a later time). Since the store instructions are kept in their program order in the store buffer, the committed entries occupy contiguous slots from the tail of the store buffer. The auxiliary variable `lsu_sb_commit_count` maintains the count of the number of committed entries. `lsu_sbi_measure_fn` is a measure function defined on the store buffer (a queue) similar to `rbi_measure_fn`. From the observations above, it follows that a valid store buffer entry (one allocated to some store instruction, indicated by the component `lsu_sb_valid`) is committed (indicated by the component `lsu_sb_commit`) if and only if its measure is less than or equal to `lsu_sb_commit_count`. And if a store buffer entry is not valid, then its measure is greater than `lsu_sb_commit_count`. These properties are formulated in [23].

5.3.2.5 Register translation table properties

The register translation table maintains one entry per register maintaining the reorder buffer index of the latest instruction modifying that register. So when the entry for a register `r` points to an instruction `rbi`, all later instructions ought to have a different destination register (if modifying the register file) and `rbi` ought to have `r` as the destination register. If the entry is zero, then none of the instructions in the reorder
buffer may write register r. These two properties are shown in \[24\]. There are similar properties for the special register file too.

\[
\text{Rtt\_non\_zero\_rf}(q:\text{state\_I}) : \text{bool} = \\
\text{FORALL} (r : \text{reg\_addr}) : \\
\ rtt(q)(r) /= 0 \text{ IMPLIES} \\
\ (\text{valid\_rb\_entry}(q,rbi) \text{ AND } \text{rb\_dest}(q)(rbi) = r \text{ AND } \text{rb\_reg\_write}(q)(rbi) \\
\text{AND} (\text{FORALL}(\text{rbi}:\text{rbiindex}): \text{(valid\_rb\_entry}(q,rbi) \text{ AND} \\
\quad \text{rbi\_measure\_fn}(q,rbi) > \text{rbi\_measure\_fn}(q,\text{rbi})) \text{ IMPLIES} \\
\quad (\text{rb\_dest}(q)(rbi) /= r \text{ OR } \text{NOT } \text{rb\_reg\_write}(q)(rbi)))
\]

\[
\text{Rtt\_zero\_rf}(q:\text{state\_I}) : \text{bool} = \\
\text{FORALL} (r : \text{reg\_addr}) : \\
\ rtt(q)(r) = 0 \text{ IMPLIES} \\
\ (\text{FORALL}(\text{rbi}:\text{rbiindex}): \text{valid\_rb\_entry}(q,rbi) \text{ IMPLIES} \\
\quad (\text{rb\_dest}(q)(rbi) /= r \text{ OR } \text{NOT } \text{rb\_reg\_write}(q)(rbi)))
\]

5.3.2.6 Properties about the feedback logic

Let rbi be an arbitrary instruction and ptr be the instruction producing one of its source values. Then ptr, when nonzero, ought to be valid, and ought to have a lower measure than rbi. Further the instructions “in between” ptr and rbi ought to have a destination different from that of ptr, and it must be the case that ptr has not yet computed a valid result. This property is shown in \[25\] for the second source operand of an instruction in issued_abs phase. And there are similar properties for its first operand and for the four operands of an instruction in issued_isu phase.

\[
\text{Abs\_rs\_ptr2\_non\_zero\_instr}(q:\text{state\_I}) : \text{bool} = \\
\text{FORALL}(\text{rbi}:\text{rbiindex}) : \\
\ (\text{valid\_rb\_entry}(q,rbi) \text{ IMPLIES} \\
\ (\text{rb\_abs\_rs}(q)(\text{rbi}) > 0 \text{ AND } \text{abs\_rs\_ptr2}(q)(\text{rb\_abs\_rs}(q)(\text{rbi})) /= 0) \text{ IMPLIES} \\
\quad \text{LET } \text{abs\_rsi} = \text{rb\_abs\_rs}(q)(\text{rbi}), \text{rbi\_ms} = \text{rbi\_measure\_fn}(q,\text{rbi}), \\
\quad \text{ptr} = \text{abs\_rs\_ptr2}(q)(\text{abs\_rsi}), \text{ptr\_ms} = \text{rbi\_measure\_fn}(q,\text{ptr}) \text{ IN} \\
\quad (\text{valid\_rb\_entry}(q,\text{ptr}) \text{ AND } \text{ptr\_ms} < \text{rbi\_ms} \text{ AND} \\
\quad (\text{NOT } \text{rb\_ready}(q)(\text{ptr}) \text{ OR } \text{rb\_exception\_raised}(q)(\text{ptr})))
\]

\[
\text{Abs\_rs\_ptr2\_non\_zero\_in\_between}(q:\text{state\_I}) : \text{bool} = \\
\text{FORALL}(\text{rbi}:\text{rbiindex}) : \\
\ (\text{valid\_rb\_entry}(q,rbi) \text{ IMPLIES} \\
\ (\text{rb\_abs\_rs}(q)(\text{rbi}) > 0 \text{ AND } \text{abs\_rs\_ptr2}(q)(\text{rb\_abs\_rs}(q)(\text{rbi})) /= 0) \text{ IMPLIES} \\
\quad \text{LET } \text{abs\_rsi} = \text{rb\_abs\_rs}(q)(\text{rbi}), \text{rbi\_ms} = \text{rbi\_measure\_fn}(q,\text{rbi}), \\
\quad \text{ptr} = \text{abs\_rs\_ptr2}(q)(\text{abs\_rsi}), \text{ptr\_ms} = \text{rbi\_measure\_fn}(q,\text{ptr}) \text{ IN} \\
\quad (\text{rb\_reg\_write}(q)(\text{ptr}) \text{ AND } \text{FORALL}(\text{rbi}:\text{rbiindex}): (\text{rbi\_ms} > \\
\quad \text{rbi\_measure\_fn}(q,\text{rbi}) \text{ AND } \text{rbi\_measure\_fn}(q,\text{rbi}) > \text{ptr\_ms}) \text{ IMPLIES} \\
\quad (\text{rb\_dest}(q)(\text{rbi}) /= \text{rb\_dest}(q)(\text{ptr}) \text{ OR } \text{NOT } \text{rb\_reg\_write}(q)(\text{rbi}))
\]
5.3.2.7 Properties about the latest instructions in the store buffer

These properties are similar to the properties about the register translation table. When an entry $\text{isu.sbi}$ in the store buffer has the latest flag set, the later entries ought to have an address different from that of $\text{isu.sbi}$. Also, among all the valid entries with a given address, one of them has the latest flag set.

5.3.2.8 Miscellaneous invariant properties

These properties pertain to the way instructions are decoded, the way circular fifo queues are implemented, etc. Some examples are:

- An instruction cannot write both the regular and the special register file.
- When an instruction is in $\text{issued.isu}$ phase, it will not write the special register file and is neither a $\text{rfeh}$ instruction nor a $\text{beq}$ instruction.
- The reorder buffer, the store buffer and the reservation stations in the LS Unit are implemented as circular fifo queues. The head, the tail and the count in these queues are always in a certain relation to one another.

Table 5.2 shows the summary of all the invariant properties discussed so far.

<table>
<thead>
<tr>
<th>The type of the invariant property</th>
<th>Number of such invariant properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exclusiveness and Exhaustiveness Instruction phase properties</td>
<td>2</td>
</tr>
<tr>
<td>Program order property in LS Unit</td>
<td>5</td>
</tr>
<tr>
<td>Properties about the committed instructions in the store buffer</td>
<td>2</td>
</tr>
<tr>
<td>Properties about the latest instructions in the store buffer</td>
<td>2</td>
</tr>
<tr>
<td>Register translation table properties</td>
<td>4</td>
</tr>
<tr>
<td>Properties about the feedback logic</td>
<td>12 (2 for each of the 6 source operands)</td>
</tr>
<tr>
<td>Miscellaneous properties</td>
<td>9</td>
</tr>
</tbody>
</table>
5.3.2.9 Other properties

We now discuss certain other properties about the implementation, formulated since they are used repeatedly in the proofs later. These properties include some simple observations on the measure functions, validity of the instructions in the buffers, properties derived from the invariant properties discussed earlier, etc. Some of these properties are:

1. If two different execution units (from the ABS Unit or the LS Unit) are busy, then their destination reorder buffer entries (the instructions to which they are assigned) are different. This implies that any given instruction may be executing on only one execution unit. A similar property holds regarding the reservation stations too.

2. If the processor is not being restarted, a valid reorder buffer entry remains valid after an implementation transition when either it is not the tail or if no instruction is retired from the reorder buffer. If a reorder buffer entry is valid after an implementation transition, then it was valid before too unless a new instruction is issued in the current cycle and the entry is the head of the reorder buffer. There are similar properties regarding the validity of entries in the store buffer and in the reservation stations in the LS Unit.

3. If \texttt{rb\_count} is not equal to zero, then the instruction at \texttt{rb\_tail} is valid. If \texttt{rb\_count} is not equal to RB, then there is no valid instruction at \texttt{rb\_head}. There are similar properties on the store buffer and the reservation stations in the LS Unit.

4. The auxiliary component \texttt{lsu\_sb\_commit\_count} is always less than \texttt{lsu\_sb\_count}, the total number of entries in the store buffer.

5. If the processor is being restarted, then there are no committed entries in the store buffer, and hence \texttt{lsu\_sb\_commit\_count} should be zero.

6. If the processor is not being restarted, then the measure of an instruction in the reorder buffer \texttt{rbi} after an implementation transition is determined as follows: If no instruction is retired, then the measure remains the same. If an instruction is retired and \texttt{rbi} is equal to the tail of the reorder buffer, then its measure will be RB (the size of the reorder buffer), otherwise it decreases by one.

7. If the processor is not being restarted and the adjusted measure of an instruction
rbi\_ms is not zero, then the instruction that corresponds to rbi\_ms in state q is the same instruction that corresponds to the adjusted measure in state I\_step(q,s,i).

5.3.3 Discharging the different proof obligations

This section explains the proofs of the different verification conditions formulated in Section 5.3.1, the proofs of the obligations due to the invariant properties, and the proofs of the other properties discussed in Section 5.3.2, and then the proof of the commutativity obligation. We provide only a high level sketch of the proofs here, glossing over some technical details. This section is organized as follows:

1. We discuss certain rewrite rules about the completion functions and some of the commonly used proof strategies.

2. We consider the proofs of the verification conditions and the lemmas discussed in Section 5.3.1.

3. We consider the proofs of the obligations due to the invariant properties and the proofs of the other properties discussed in Section 5.3.2.

4. We consider the proofs of the commutativity obligation and the projection obligation.

5.3.3.1 Rewrite rules and proof strategies

The completion functions do not modify any of the nonobservable components in the implementation machine. We prove a lemma that captures this property and then instruct PVS to use the lemma as a rewrite rule. For each nonobservable and for each unfinished instruction rbi in the reorder buffer, we generate a rewrite rule that states that the nonobservable is unaffected by completing the instructions till rbi. We generate a similar rule for the committed store instructions in the store buffer. For example, the rewrite rules for abs\_rs\_valid are shown in \[26\]. All these rules can be generated automatically once the completion functions are specified, and their proofs are all straightforward.

```plaintext
rewrite_rb_abs_rs_valid: LEMMA
   FORALL (q:state_I,rbi\_ms:rbindex_p):
      abs\_rs\_valid(proj_1(Complete\_Squash\_rest?_till(q,rbi\_ms))) = abs\_rs\_valid(q)

rewrite_sb_abs_rs_valid: LEMMA
   FORALL (q:state_I,lsu\_sbi\_ms:LSU_sbiindex_p):
      abs\_rs\_valid(Complete\_committed\_in\_sb\_till(q,lsu\_sbi\_ms)) = abs\_rs\_valid(q)
```
Other lemmas used as rewrite rules include "record accessor" lemmas about the implementation and the specification machine states, and the lemmas about select/assign operations for the register file and read/write operations for the data memory (see Appendix B). In addition, the definitions from the implementation and the specification machines, the abstraction function related definitions, and some of the properties proved about the implementation machine are used as rewrite rules too. We then define strategies that lets us instruct the prover to use these lemmas/definitions as rewrite rules, and certain other simplification strategies such as the case-analysis strategy. See Appendix C for more details.

5.3.3.2 Proving the verification conditions

The verification conditions when the processor is being restarted: The proofs of these verification conditions are straightforward, involving expanding the definitions of the functions/predicates appearing in the verification conditions (directly or indirectly), using the necessary properties/invariants and simplifying the resulting expressions. Consider the proof of the first verification condition Restarting rf.rb_tail_VC shown in 9. From a property discussed in Section 5.3.2 ("Other properties", numbered 5), there are no committed instructions in the store buffer. So the left hand side simplifies to rf(Complete_instr(q,rb_tail(q),FALSE)), as Completecommitted sb_till will be vacuous on the left hand side. 7 When the processor is being restarted, the instruction at the tail of the reorder buffer is in writtenback phase, so the left hand side further simplifies to rf(Action_writtenback(q,rb_tail(q))). Proof now proceeds by expanding out the remaining function definitions on both sides and simplifying the resulting expression using the case-analysis strategy. The proofs of the other verification conditions are similar.

The proofs of the lemmas generated due to the instruction phase transitions shown Figure 5.2: The proofs of these lemmas are straightforward too, involving using the necessary properties/invariants and simplifying the resulting expressions. As an example, consider the proof of the second conjunct of abs_executed_to_writtenback shown in 11. From a property discussed in Section 5.3.2 ("Other properties", numbered 2), it follows that

---

7In describing the proofs in this dissertation, we refer to the "main" goal we are proving in a verification condition (rf(…) = rf(…) or dsem(…) = dsem(…) etc) as the consequent. When we just say the "left hand side" or the "right hand side" in a proof without qualifying it further, we are referring to this consequent.
rbi remains valid after an implementation transition unless it is the tail of the reorder buffer and reorder buffer is retiring rbi. However this requires rbi to be in the written-back phase, but it is in executed_abs phase. And according to Instruction_exclusive invariant, it cannot be in two phases simultaneously. So rbi remains valid after an implementation transition.

The proof of rf_sp_rf_mode_reg_dmem_Squash_rest?_till_remains_same_VC: This verification condition is decomposed into 14 obligations as discussed in Section 5.3.1.2, and we discuss the proof of one these obligations (absissued_remains_induction in [12]). Its proof is further decomposed into five obligations, one each for the five conjuncts in the definition of rf_sp_rf_mode_reg_dmem_Squash_rest?_till_remains_same in [10]. Consider the proof for the observable rf. (Only a high level sketch is given here; more details of the proof are provided in Appendix D.) We expand the definitions of Complete_till and then that of Complete_Squash_rest?_till on both sides of the consequent. It follows from absissued_remains lemma (not shown in Section 5.3.1.2 but similar to absexecuted_remains shown in [11]) that rbi remains in issued_abs phase and remains valid after an implementation transition, and that the adjusted measure is nonzero. Since rbi_ms is nonzero and since the adjusted measure is nonzero, from the definition of Complete_Squash_rest?_till, the consequent simplifies to the expression8 shown in [27]. That is, we are required to show that the effect of completing rbi (after completing the instructions ahead of it) is the same on rf in both states q and I_step(q,s,i).

rf(Complete_instr(Complete_till(q,rbi_ms-1), rbi, Squash_rest?_till(q,rbi_ms-1)))

= rf(Complete_instr(Complete_till(I_step(q,s,i),rbi_measure_adjust(q,s,rbi_ms)-1), rbi, Squash_rest?_till(I_step(q,s,i),rbi_measure_adjust(q,s,rbi_ms)-1)))

Consider the case when rbi_ms is one, that is, we are considering the instruction at the tail of the reorder buffer. This corresponds to the basis case of the induction argument and rb_induction_hypothesis will be vacuous. From Instruction_exclusive invariant, rbi can be only in one phase at any time, and in this case, it is in issued_abs phase in

---

8In the actual sequent in the PVS proof, Complete_till and Squash_rest?_till appear expanded, and q, rbi_ms etc appear respectively as q!1, rbi_ms!1 etc. It is shown differently here for better readability.
both states q and I\_step(q,s,i). When rbi\_ms is one, there are no earlier instructions in the reorder buffer, and Squash\_rest?\_till is FALSE on both sides of the consequent. From the definition of Complete\_instr in [2] the consequent simplifies further to the expression shown in [28].

```plaintext
rf(Action\_issued\_abs(Complete\_committed\_in\_sb\_till(q,lsu\_sb\_commit\_count(q)),
          rb\_tail(q)))
  =
rf(Action\_issued\_abs(Complete\_committed\_in\_sb\_till(I\_step(q,s,i),
          lsu\_sb\_commit\_count(I\_step(q,s,i))),
          rb\_tail(q)))
```

Proof now proceeds by expanding the function Action\_issued\_abs, expanding the implementation machine terms that occur on the right hand side of the consequent and simplifying the resulting expression. In a correct implementation, it ought to simplify to TRUE.

Now consider the case when rbi\_ms is not equal to 1. From rb\_induction\_hypothesis, it follows that the Squash\_rest?\_till predicates that appear on both sides of the consequent have the same truth value, that is, either rbi is killed in both states q and I\_step(q,s,i) or it is not. When rbi is killed, the consequent simplifies to the expression shown in [29], which again simplifies to TRUE immediately from the induction hypothesis.

```plaintext
rf(Complete\_till(q,rbi\_ms-1)) =
rf(Complete\_till(I\_step(q,s,i),rbi\_measure\_adjust(q,s,rbi\_ms-1)))
```

When rbi is not killed, using the Instruction\_exclusive invariant as in the basis case, expanding and simplifying the Complete\_instr function on both sides, the consequent simplifies to the expression shown in [30].

```plaintext
rf(Action\_issued(Complete\_till(q,rbi\_ms-1),
          rbi))
  =
rf(Action\_issued(Complete\_till(I\_step(q,s,i),rbi\_measure\_adjust(q,s,rbi\_ms-1)),
          rbi))
```

Proof now proceeds by expanding the function Action\_issued\_abs, expanding the implementation machine terms that occur on the right hand side of the consequent and simplifying the resulting expression. In a correct implementation, it ought to simplify to
TRUE. If Action\_issued refers to any of the observable components rf, sp\_rf, mode\_reg or dmem, those values are the same on both sides of the consequent by induction hypothesis.

We now consider the proof of another conjunct of abs\_issued\_remains\_induction, namely the one corresponding to the Squash\_rest?\_till predicate. Following steps similar to that used in the proof for the observable rf, the consequent simplifies to the expression shown in \[31\].

\[
\begin{align*}
\text{Squash\_rest?\_instr}(\text{Complete\_till}(q,\text{rbi\_ms-1}),\text{rbi}) \\
\quad \text{OR Squash\_rest?\_till}(q,\text{rbi\_ms-1}) \\
= \\
\text{Squash\_rest?\_instr}(\text{Complete\_till}(\text{I\_step}(q,s,i),\text{rbi\_measure\_adjust}(q,s,\text{rbi\_ms})-1),\text{rbi}) \\
\quad \text{OR Squash\_rest?\_till}(\text{I\_step}(q,s,i),\text{rbi\_measure\_adjust}(q,s,\text{rbi\_ms})-1)
\end{align*}
\]

Proof proceeds as in the earlier illustration by considering the basis and the induction cases, expanding the Squash\_rest?\_issued\_abs definitions that appear on both sides, and simplifying the resulting expressions. In the induction case, the Squash\_rest?\_till predicates that appear on both sides of the consequent have the same truth value by the induction hypothesis.

The proofs of the other 12 obligations in the decomposition except the one corresponding to the case where an instruction in the written\_back phase is being retired follow the same pattern discussed above. We make some observations regarding these proofs:

1. Recall how the source value is specified for an instruction in issued\_abs phase. For example, the second source operand for such an instruction is specified by abs\_src2\_value shown in \[5\]. Now, in the proof of abs\_issued\_remains\_induction discussed above, rbi is in issued\_abs phase in both states q and I\_step(q,s,i). We use the verification conditions regarding the correctness of the source operands to relate them on the two sides of the consequent. In the proof above, it can be inferred that the second source operand is the same on both sides of the consequent by using abs\_src2\_value\_same\_lemma. Similarly, the verification conditions regarding the correctness of the other source operands are used in the proofs of the appropriate obligations.

2. In the proof of abs\_issued\_remains\_induction, rbi is in issued\_abs phase in both states q and I\_step(q,s,i). But we are actually talking of completing rbi in a state where the instructions ahead of it are completed, and not just q or I\_step(q,s,i). For example, on the left hand side of the consequent, we are talking of com-
pleting rbi in the state Complete_till(q,rbi_ms-1). Now rbi is in issued_abs phase in Complete_till(q,rbi_ms-1) too, and this follows from the fact that Instr_issued_abs? predicate is defined using nonobservable components only and the nonobservable components are not modified by the completion functions. By a similar observation, rbi is in the issued_abs phase on the right hand side of the consequent too.

3. The case where an instruction in writtenback phase is being retired applies only to the instruction at the tail of the reorder buffer, as no other instruction can be retired. So the proof of the obligation corresponding to this scenario is simpler with no induction case as in the proof described earlier.

4. We need to use the verification condition store_buffer.dmem_remains_same_VC due to the committed instructions in the store buffer while considering the proof for the observable dmem in all the 14 obligations in the decomposition.

The proofs of the verification conditions due to the feedback logic and regarding the correctness of source operands: Consider the proof of abs_src2_value_feedback_VC shown in [14]. This scenario is depicted in Figure 5.3. rbi is a valid instruction in the reorder buffer in issued_abs phase and let rs (= rb.abs.rs) be the reservation station allocated to it. ptr is the instruction producing the second source operand for rbi and the component abs_rs_ptr2 in rs points to ptr. Since abs_rs_ptr2 is nonzero and abs_rs_ptr2' is zero, then in a correct implementation, ptr will complete execution on some execution unit eu (in either the ABS Unit or the LS Unit) and forward its result to all the instructions waiting for it (in particular, to rbi) in the current implementation transition. So abs_rs_val2' ought to be equal to this forwarded result value. Now consider the right hand side of the consequent. From the invariant properties shown in [25], the instructions in between rbi and ptr (shown shaded) either do not write the register file or have a destination different from that of the destination of ptr. (The destination register of ptr is the same as the second source register of rbi.) So, as far as reading the destination register of ptr is concerned (refer to the definition of abs_src2_value in [5]), the instructions in between rbi and ptr do not matter. The instruction ptr is not being killed by the earlier instructions, otherwise Squash_rest_.till(q,rbi_ms-1) will be TRUE. Since ptr is in either executed_abs phase or in executed_lsu phase, the right hand side of the consequent simplifies to the value as given by Action_executed_abs
or `Action_executed_lsu` for `ptr`, and this value is the result value in the appropriate execution unit. So the consequent ought to simplify to `TRUE`.

We now discuss the proof of `abs_src2_value_same_VC`. From `abs_issued_remains` lemma, `rb_abs_rs'` is equal to `rb_abs_rs`. Referring to the definition of `abs_src2_value` in 51, there are three possible cases. When `abs_rs_ptr2` is zero, `abs_rs_ptr2'` ought to be zero too and the proof reduces to showing that `abs_rs_val2` and `abs_rs_val2'` are the same, and this ought to hold in a correct implementation. (The source operand cannot change when there is no dependency.) When `abs_rs_ptr2` is nonzero, `abs_rs_ptr2'` may or may not be zero. If `abs_rs_ptr2'` is zero, the proof follows from the verification condition `abs_src2_value_feedback_VC`. If `abs_rs_ptr2'` is nonzero, then we show that `abs_rs_ptr2'` is equal to `abs_rs_ptr2`, and the proof follows by using the induction hypothesis.

The proof of `mode_reg_unchanged_VC` shown in 13 is by a straightforward induction on `rbi_ms`. This verification condition involves only the definitions introduced by us while specifying the completion function, and does not involve any implementation transition. 

**Other verification conditions discussed in Section 5.3.1:** The proofs of the remaining verification conditions and lemmas discussed in Section 5.3.1 are either straightforward or similar to the proof of a verification condition discussed before, and hence are skipped.
5.3.3.3 Proving the invariants and the other properties

The invariant properties discussed in Section 5.3.2 define a superset of the reachable states of the implementation machine. The other properties discussed in that section follow either from these invariants or from other observations about the implementation machine.

We are required to prove two obligations regarding these invariant properties:

- That they hold in the initial states of the implementation machine.
- That they hold in a state reached after an implementation transition from a state where they hold.

We have no uniform strategy applicable to all these obligations, however their proofs are all straightforward involving using the necessary invariant and other properties about the implementation machine (appropriately instantiated), and simplifying using rewriting and the simplification strategies discussed in Appendix C. Similarly there is no uniform strategy for the other properties mentioned in Section 5.3.2.9. Their proofs involve using the necessary invariant properties and the other properties proved earlier, and simplifying using the strategies discussed in Appendix C.

5.3.3.4 Proving the commutativity obligation

As explained in Chapter 3, the structure of the synchronization function leads to different cases in the proof of the commutativity obligation.

We first consider the case when the processor is being restarted. We explain the proof for the observable \(rf\), and the proofs for the other observable components are similar. Since the synchronization function returns zero in this case, the commutativity obligation (for \(rf\)) is of the form shown in \[\text{Table 32}\].

<table>
<thead>
<tr>
<th>\text{rf_commutes_restarting_case: LEMMA}</th>
<th>\text{32}</th>
</tr>
</thead>
<tbody>
<tr>
<td>restart_proc(q,s)\text{ IMPLIES}</td>
<td></td>
</tr>
<tr>
<td>(rf(\text{ABS}(I_step(q,s,i)))) = rf(A_step_new(\text{ABS}(q),0)))</td>
<td></td>
</tr>
</tbody>
</table>

- If the processor is being restarted in the current cycle, then it discards all the executing instructions in the reorder buffer, and sets \text{rb\_count} and \text{lsu\_sb\_commit\_count} to zeros (i.e. \text{rb\_count'} and \text{lsu\_sb\_commit\_count'} are zeros). So \text{Complete\_till} is vacuous on the left hand side, and it simplifies to \(rf(I\_step(q,s,i))\).
- The Squash_rest?_till predicate is TRUE for the instruction at the tail of the reorder buffer from Restarting_Squash_rest?_till_rb_tail_VC. From the definition of Complete_Squash_rest?_till in [7], it follows that the kill? argument (y in [7]) is TRUE for all the remaining instructions in the reorder buffer. So none of the executing instructions in the reorder buffer other than the one at the tail affects the observable components, and hence the right hand side simplifies to rf(Complete_till(q,1)).

- The proof immediately follows by using Restarting_rf_rb_tail_VC.

Now consider the case when one of the instructions in the processor is causing all the subsequent instructions to be squashed, that is, Squash_rest?_till(q,rb_count(q)) is TRUE (and restart_proc? is FALSE). The synchronization function returns zero in this case too, and the commutativity obligation (for rf) is of the form shown in [33].

```
rf commutes_Squash_rest?_till_case: LEMMA
  (NOT restart_proc?(q,s) AND Squash_rest?_till(q,rb_count(q)))
  IMPLIES
  rf(ABS(I_step(q,s,i))) = rf(I_step_new(ABS(q),0))
```

Since the Squash_rest?_till predicate is TRUE for the latest instruction in the reorder buffer in state q, it is TRUE of the corresponding instruction in state I_step(q,s,i) by the fifth conjunct of rf_sp_rf_mode_reg_dmem_Squash_rest?_till_remains_same_VC. This implies the kill? argument for the new instruction fetched, if any, is TRUE in state I_step(q,s,i), and hence it has no effect on rf. The proof now follows by the first conjunct of rf_sp_rf_mode_reg_dmem_Squash_rest?_till_remains_same_VC. The proof for the other observable components is similar except that the proof for pc uses pc_remains_same_VC1, and the proof for imem is straightforward.

Now consider the case when no new instructions are issued into the pipeline (and Squash_rest?_till(q,rb_count(q)) and restart_proc? are FALSE). The proofs for the four observable components rf, sp_rf, mode_reg and dmem directly follow from the first four conjuncts of rf_sp_rf_mode_reg_dmem_Squash_rest?_till_remains_same_VC. The proof for pc follows from pc_remains_same_VC2, and the proof for imem is straightforward. Finally the proofs for the case when a new instruction is issued (and the first three conditions of the synchronization function are FALSE) follow directly from the verification conditions about the new instructions.
CHAPTER 6

PROCESSORS WITH OUT-OF-ORDER COMPLETION

The previous chapters illustrated the application of our approach to in-order completion processor examples, without and with out-of-order execution. Those ideas, however, cannot be directly applied to out-of-order completion machines. These machines allow the executing instructions to commit their results to some or all the observable components out of program order.\footnote{It is hard to support speculative execution and precise exceptions in the presence of out-of-order completion. However, out-of-order completion mechanism can still be used in the implementation of specialized execution units such as those processing arithmetic instructions only.} Since the instructions may commit their results out of program order, it is not possible to relate the effect on the observable components of instructions completing one at a time in their program order, as is done in the earlier chapters. Therefore, defining the abstraction function by composing the completion functions of the unfinished instructions in their program order will not necessarily lead to a similar decomposition as in the case of the examples considered in the earlier chapters. Also, given an unfinished instruction, it is not possible to provide an "abstract" state where (only) the instructions ahead of it are completed. This fact is useful in expressing the correctness of the feedback logic.

This chapter illustrates how to modify the basic completion functions approach appropriately to handle out-of-order completion machines. The amount of modification necessary depends on the extent to which the machine allows out-of-order completion of instructions. This is illustrated on two examples. EX3.1 is an example processor with limited out-of-order execution capability. It allows a certain arithmetic instruction to bypass a certain other arithmetic instruction when their destination registers are different. EX3.2 is an example processor implementing Tomasulo's algorithm [54] without a reorder buffer. In its implementation, any instruction may execute when its operands are ready,
and commit its result to the observable components if not “overridden” by a subsequent instruction.

6.1 Application to EX3.1

6.1.1 Processor details

An implementation model of EX3.1 is shown in Figure 6.1. The implementation has six sets of pipeline latches, shown by the patterned rectangular boxes in the figure. It has three execution units: an adder, a load/store unit and a multiplier, all sharing the EX/WB set of latches. An add instruction takes one cycle to execute, a load instruction takes two cycles and a mult instruction takes three cycles. Since the EX/WB set of latches is shared between these three execution units (i.e., they all write their results into EX/WB set of latches), this situation represents a structural hazard and is resolved by the issue logic by not issuing instructions such that two may simultaneously attempt to enter the write-back stage. This implies that only one of the following three cases may hold at any time: a valid mult instruction is present in ML2/ML3 set of latches, a valid load is present in LD1/LD2 set of latches, and a valid add about to be issued is present in DC/EX set of latches. The processor has no reorder buffer, hence instructions retire immediately after execution. So, an add instruction, issued immediately after a mult instruction, can bypass it and complete before it updating the register file out of program order. However the processor issues such an add instruction only if its destination register is different from that of the mult instruction issued earlier to ensure that there is no write-after-write hazard between these two instructions. (The situation of an add bypassing either a load instruction or a mult instruction issued two cycles earlier cannot occur due to the absence of structural hazards). The scheduling registers keep track of the current instructions executing in the three execution units. The processor has two other instructions: a store instruction and a conditional branch instruction beqz. The issue logic issues the instructions one at a time when its operands are ready.

6.1.2 Constructing the abstraction function

Recall that the first step in constructing the abstraction function in our approach is to determine the unfinished instructions in the pipeline and their program order. The implementation can have six unfinished instructions at any time in the six sets of pipeline latches. However, unlike in the examples considered in the earlier chapters, their program order is not directly available. In particular, the ordering among the instructions EX/WB
and ML2/ML3 is not clear. For instance, both could be mult instructions with EX/WB earlier than ML2/ML3 in the program order, or EX/WB could be an add instruction that has bypassed ML2/ML3, a mult instruction occurring earlier in the program order. However, even though an add instruction can bypass a mult instruction and update the register file out of program order, the (correctly working) machine allows this to happen only if their destination registers are different. So though the register file contents when such an add is completed does not correspond to those of a specification machine state, it does so when both the add and the mult are completed.

In the current example, we choose to complete the unfinished instructions in the order: EX/WB, ML2/ML3, ML1/ML2, LD1/LD2, DC/EX and IF/DC. This corresponds to the program order of these instructions under all circumstances except when an add instruction is issued. Then, in the next implementation state, ML2/ML3 occurs earlier in the program order than EX/WB. This situation needs to be suitably accounted for in generating the

\footnote{ML1/ML2 and LD1/LD2 can not be both valid simultaneously since the issue logic issues only one instruction at a time. So these two instructions can be interchanged in the order shown. Similarly it is possible to justify other instruction orderings.}
verification conditions (discussed below).

The rest of the process of specifying the completion functions and the squashing predicates, and that of constructing the abstraction function is similar to those of the examples described in the earlier chapters. Those details are skipped here (and it is assumed that various functions and predicates are named appropriately).

### 6.1.3 Proof decomposition

If the instruction DC/EX is issued in the current implementation transition (the signal dc_ex_stalled? is FALSE), the effect of completing all the instructions till DC/EX on the register file ought to be the same in the present and the next states, whether or not DC/EX bypasses an earlier issued instruction. This verification condition VC5.rf_case1 is shown in [1]. The proof of this verification condition is explained in detail, using the verification conditions due to the earlier instructions. The verification conditions for rf due to the subsequent instructions, the verification conditions for the other observable components and the squashing predicates, and the proof of the commutativity obligation are all similar to the examples considered in Chapter 4.

```
VC5.rf_case1: CLAIM
    NOT dc_ex_stalled?(q) IMPLIES
    rf(Complete_till_LD1_LD2(I_step(q,i))) = rf(Complete_till_DC_EX(q))
```

Consider the instruction EX/WB. In the current implementation transition, this instruction updates the register file and gets retired. So the contents of the register file in I_step(q,i) ought to be the same as after completing EX/WB, that is, in state Complete_till_EX WB. This verification condition VC1.rf is depicted in Figure 6.2.

Now what is rf(Complete_till_EX WB(I_step(q,i))) equal to? There are four possible cases depending on the instruction about to enter into EX/WB phase:

1. A valid mult instruction in ML2/ML3 set of latches.
3. A valid add instruction about to issued in DC/EX set of latches.
4. None of these three possibilities.

That these cases are mutually exclusive follows from the fact that structural hazard is resolved properly by the issue logic. Now instead of generating the verification conditions
Figure 6.2. Verification conditions for rf due to the first four instructions in three of the cases (m, l, o)
individually for the instructions ML2/ML3, ML1/ML2 and LD1/LD2, it is convenient to generate them depending on the cases listed above. The verification conditions for three of the cases listed above (m, 1, o) are shown in Figure 6.2. (The suffixes m*, 1* and o* identify the verification conditions of the corresponding cases.)

Consider the case when there is a valid mult instruction in the ML2/ML3 set of latches (i.e., case m). In the current implementation transition, the instruction ML2/ML3 goes into EX/WB phase, and hence Complete_EX WB can be used to complete it in I_step(q,i). So the verification condition VC_rf.m1 ought to hold. Similarly the verification condition VC_rf.m2 can be justified. Now when the instruction ML2/ML3 is valid, there can not be a valid load instruction in the LD1/LD2 set of latches, so Complete_LD1_LD2 has no effect on rf on the specification side, so VC_rf.m3 ought to hold.

Consider the case when a valid load instruction is present in the LD1/LD2 set of latches (i.e., case l). Now instruction ML2/ML3 can not be valid. Neither is instruction ML1/ML2, as instructions are issued one at a time. So CompleteML2_ML3 and CompleteML1_ML2 have no effect on rf on the specification side, so VC_rf.l1 should hold. Since the instruction LD1/LD2 goes to EX/WB phase after an implementation transition, VC_rf.l2 should hold. Since the instruction ML1/ML2 is invalid, after I_step, the instruction ML2/ML3 is invalid too, so CompleteML2_ML3 should have no effect on rf on the implementation side, and so VC_rf.l3 should hold. The justification for the verification conditions for the fourth case (i.e., case o) is similar.

The interesting case is the remaining scenario of an add instruction being issued from the DC/EX set of latches. The issued add instruction will be in EX/WB phase in I_step(q,i). The verification conditions in this case are shown in Figure 6.2. When an add instruction is being issued from the DC/EX set of latches, there can not be a valid instruction in the ML2/ML3 set of latches, and hence VC_rf.i1 ought to hold. The instruction ML1/ML2 goes to ML2/ML3 phase after an implementation transition, and hence VC_rf.i2 ought to hold. Observe that the effect of EX/WB instruction is not yet considered on the implementation side and that corresponds to considering the effect of DC/EX instruction on the specification side. Since LD1/LD2 set of latches has no valid load instruction when an add is being issued, VC_rf.i3 ought to hold. Finally, since the add instruction in DC/EX set of latches goes to EX/WB phase after an implementation transition, VC_rf.i4 ought to hold.
VC_rf_i1: CLAIM

issue_add?(q) IMPLIES
rf(I_step(q,i)) = rf(Complete_till_ML2_ML3(q))

VC_rf_i2: CLAIM

issue_add?(q) IMPLIES
rf(Complete_ML2_ML3(I_step(q,i))) = rf(Complete_till_ML1_ML2(q))

VC_rf_i3: CLAIM

issue_add?(q) IMPLIES
rf(Complete_ML2_ML3(I_step(q,i))) = rf(Complete_till_LD1_LD2(q))

VC_rf_i4: CLAIM

issue_add?(q) IMPLIES
rf(Complete_EX_WR(Complete_ML2_ML3(I_step(q,i)))) =
rf(Complete_till_DC_EX(q))

Observe that on the implementation side of VC_rf_i4, the completion functions corresponding to EX/WB and ML2/ML3 instructions are in the opposite order to that required by the abstraction function. However, whenever the processor issues an add instruction that bypasses a mult instruction issued earlier, their destination registers are different. This fact can be used to prove a reordering lemma shown in 3 that allows the order of the two completion functions on the implementation side in VC_rf_i4 to be switched.

reordering lemma: LEMMA

issue_add?(q) IMPLIES
rf(Complete_till_ML2_ML3(I_step(q,i))) =
rf(Complete_EX_WR(Complete_ML2_ML3(I_step(q,i))))

Now consider the proof of VC5_rf_case1. When the instruction DC/EX is not stalled, the following scenarios are possible:

- DC/EX is an add instruction. In that case, there is no valid instruction in ML1/ML2 and LD1/LD2 sets of latches in I_step(q,i), and the proof can be accomplished using reordering lemma and VC_rf_i4.

- DC/EX is a mult instruction. It goes to ML1/ML2 phase after an implementation transition and there is no valid instruction in LD1/LD2 set of latches in I_step(q,i). The proof is accomplished by showing that the DC/EX instruction has the same effect on register file as ML1/ML2 instruction in I_step(q,i) in the three cases m, l and o using respectively VC_rf_m3, VC_rf_l3 and VC_rf_o3.

- DC/EX is a load instruction. It goes to LD1/LD2 phase after an implementation transition and there is no valid instruction in the ML1/ML2 set of latches in I_step(q,i).
The proof is accomplished by showing that the DC/EX instruction has the same effect on register file as LD1/LD2 instruction in \( I_{\text{step}}(q, i) \) in the three cases \( m, 1 \) and \( o \) using respectively \( VC_{rf,m3}, VC_{rf,13} \) and \( VC_{rf,o3} \).

- In the remaining cases (say, DC/EX is a beqz instruction), neither does DC/EX affect the register file in \( q \) nor does the instructions ML1/ML2 and LD1/LD2 in \( I_{\text{step}}(q, i) \). The proof is accomplished in the three cases \( m, 1 \) and \( o \) using respectively \( VC_{rf,m3}, VC_{rf,13} \) and \( VC_{rf,o3} \).

The remaining details in the the proof of the commutativity obligation are similar to those of the examples considered in Chapter 4 and is not discussed here.

### 6.2 Application to EX3.2

#### 6.2.1 Processor details

A block diagram model of EX3.2 implementing Tomasulo's algorithm without a reorder buffer is shown in Figure 6.3. The model has RS reservation stations and EU execution units; both RS and EU are uninterpreted positive constants. Every instruction in the processor has a reservation station allocated, and it records some information about the instruction waiting to be dispatched onto an execution unit. A register translation table (RTT), with one entry per register, maintains the identity of the latest pending instruction writing a particular register. If the entry for a register is zero, then that register is not being written by any of the executing instructions. Since every instruction in the processor has a reservation station allocated, its index is used as a tag to identify the instruction. A scheduler controls the movement of the instructions through the execution pipeline (such as being dispatched, executed etc.) and its behavior is modeled by a collection of axioms. Instructions are fetched from the instruction memory using the program counter which then is incremented. The implementation also takes a no_op? input that suppresses an instruction fetch when asserted. Only one instruction type, namely alu, is supported.

An instruction is issued by allocating a free reservation station for it (sch\_rs\_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. \texttt{sch\_di}\? and \texttt{sch\_di} outputs from the scheduler (each a EU-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 \textit{executed} after a non-deterministic amount of time as determined by the scheduler output \texttt{sch\_ex}\? (a EU-wide vector). At a time determined by the \texttt{sch\_wb}\? output of the scheduler (a EU-wide vector), 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.

\textbf{Figure 6.3.} Implementation model of EX3.2
6.2.2 Why is the basic completion functions approach hard to apply?

The basic completion functions approach is hard to use in this example because of two reasons. First, the processor allows the instructions to commit out of program order. So the processor does not maintain the executing instructions in their program order. Nor is the program order (if available) the appropriate order to use in constructing the abstraction function, as such a construction of the abstraction function will not lead to a similar decomposition of the proof as in the basic approach. Second, not every instruction in the processor updates the register file, when it is done executing. The implementation maintains the identity of the latest instruction writing a particular register, and only these instructions modify the register file when done executing. Those instructions issued earlier and not the latest ones to write their respective destinations, on completing their execution, only forward the results to the other waiting instructions (if necessary). In the basic approach, the completion function for an unfinished instruction is defined to directly update the observable components.

So the basic completion functions approach is modified appropriately to be applicable in this scenario. Instead of defining the completion function to directly update the observable components, it is defined to return the value an instruction computes in the various phases. The completion function for a given unfinished instruction recursively completes the instructions it is dependent on to obtain its source operand values. The abstraction function is defined to assign to a register the value computed by the latest instruction writing that register. It is now illustrated how this modified approach leads to a similar decomposition of the overall proof of correctness.

6.2.3 Constructing the abstraction function

Every unfinished instruction in the processor has a reservation station allocated to it, and we identify an instruction by its reservation station index (i.e., instruction rsi means instruction at reservation station index rsi). An unfinished instruction in the processor can be in one of the three following phases: issued, dispatched or executed. Once written back, the instruction is no longer present in the processor. We formulate predicates that describe an instruction in each of these phases and functions that specify the value an instruction computes in each of these phases. (There is no squashing predicate as there is no notion of squashing in this implementation. It is very difficult to allow for precise exceptions and speculative execution in the presence of out-of-order completion
of instructions [33].) The definition of the completion function, applicable to all the
unfinished instructions in the processor, is shown in 4.

Complete_instr(q:Reach_state_I, rsi:rsiindex): RECURSIVE value =
   IF Instr_executed?(q, rsi) THEN Action_executed(q, rsi)
   ELSIF Instr_dispatched?(q, rsi) THEN Action_dispatched(q, rsi)
   ELSIF Instr_issued?(q, rsi) THEN
      Action_issued(q, rsi,
         IF rs_ptr1(q)(rsi) = 0 THEN rs_val1(q)(rsi) % First source operand.
         ELSE Complete_instr(q, rs_ptr1(q)(rsi)) ENDIF,
         IF rs_ptr2(q)(rsi) = 0 THEN rs_val2(q)(rsi) % Second source operand.
         ELSE Complete_instr(q, rs_ptr2(q)(rsi)) ENDIF)
   ELSE default_value % This should not happen.
   ENDIF
MEASURE rs_instr_num(q)(rsi)

In this implementation, when an instruction is in the *executed* phase, the result value is
available in the eu_result component of the execution unit. Therefore *Action_executed*
returns this value. We specify *Action_dispatched* and *Action_issued* along the same
lines. The main difference is in the way we specify the source operands for an instruction
in the *issued* phase since they may not be ready. When rs_ptr1 is zero, the first source
operand is ready and its value is available in rs_val1, otherwise its value is obtained by
recursively completing the instruction it is waiting on (rs_ptr1 points to that instruction).
Similarly the second source operand is specified.

To facilitate the above formulation of the completion function, we add three auxiliary
components to the implementation. The first one maintains the index of the execution
unit an instruction is dispatched to. Since the completion function definition is recursive,
we need to provide a measure function to show that the function is well-defined; the
other two auxiliary components are for this purpose. We must prove that instructions
producing the source operand values for a given instruction rsi have a lower measure
than the measure of 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 component.

Now the abstraction function is defined to update every register with the value ob-
tained by completing the latest instruction writing that register as shown in 5. The
register translation table maintains the identity of the latest instruction writing a partic-
ular register.
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. Its definition is shown in [5] too.

### 6.2.4 Proof decomposition

This section is organized as follows: We first discuss the formulation of a key verification condition that relates the value an instruction computes in the present and the next implementation states. We explain how the proof of this verification condition is decomposed. We then discuss the proof of the commutativity obligation and the invariant properties needed.

#### 6.2.4.1 A key verification condition and its proof

Consider an arbitrary instruction rsi. An instruction is valid as long as it has not finished its execution and written back its result, and this is indicated by the implementation component rs_valid. We claim that the value an instruction computes (as given by Complete_instr) is the same in states q and I_step(q,s,i), as long as the instruction is valid in these states. This verification condition is shown as same_result_VC in [6]. We prove this by an induction on rsi (induction with a measure function as explained below).

```
same_result_VC: CLAIM
FORALL(rsi:rsiindex):
(rsi_valid(q)(rsi) AND rs_valid(I_step(q,s,i))(rsi))
IMPLIES
Complete_instr(q,rsi) = Complete_instr(I_step(q,s,i),rsi)
```

We decompose the proof of this verification condition into different cases based on how an instruction makes a transition from its present phase to its next phase. The instruction phase transitions are shown in Figure 6.4 where the predicates labeling the arcs define the conditions under which an instruction changes phase. For example, we
Figure 6.4. Various phases an instruction can be in and transitions between them, I: issued, D: dispatched, E: executed

identify the predicate Dispatch?(q,s,rsi) which takes the instruction rsi from issued phase to dispatched phase. In this implementation, this predicate is TRUE when there is an execution unit for which sched output from the scheduler is TRUE and the sched output is equal to rsi. Similarly other “transition” predicates are defined. The figure also shows the transition for a new instruction entering the processor.

Having defined these predicates, we prove that they indeed cause instructions to take the transitions shown. Consider a valid instruction rsi in issued phase. We prove that if Dispatch? predicate is TRUE, then after an implementation transition, rsi will be in dispatched phase and remains valid. (This is shown as a lemma in 7.) Otherwise (if Dispatch? is FALSE), we prove that rsi remains in issued phase in I step(q,s,i) and remains valid. The lemmas for the other transitions are similar except for the case when an instruction rsi in the executed phase is written back. In this case, rsi will not be valid in I step(q,s,i).

issued_to_dispatched: LEMMA
FORALL(rsi:rsindex):
 (rs_valid(q)(rsi) AND Instr_issued?(q,rsi) AND Dispatch?(q,s,rsi))
 IMPLIES
 (Instr_dispatched?(I step(q,s,i),rsi) AND rs_valid(I step(q,s,i))(rsi))

Now we come back to the details of the proof of same_result_YC. In proving this verification condition for an instruction rsi, we need to assume that it holds for the two instructions producing the source operand values for rsi. So we do an induction on rsi with rs_instr_num as the measure function. As explained earlier in Section 6.2.3, instructions producing the source values (rs_ptr1 and rs_ptr2 when nonzero) have a lower measure than the measure of rsi. We do a case analysis on the possible phase rsi can be in, and whether or not it makes a transition to its next phase. Assume that the instruction rsi is in dispatched phase. Now there can be two cases depending on whether Execute? is TRUE or FALSE. The proof obligation for the second case is shown in 8. We
have similar proof obligations for \( rsi \) being in other phases. In all, the proof decomposes into five obligations. (When an instruction in the executed phase is written back, it is not valid in the next implementation state, and there is no obligation for this case.) 8 also shows the induction hypothesis induction_hypothesis.

<table>
<thead>
<tr>
<th>induction_hypothesis(q:Reach_state_I,s:schedule,i:inputs_type, rsi:rsindex): bool =</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORALL (y:rsindex): rs_instr_num(q)(y) &lt; rs_instr_num(q)(rsi)</td>
</tr>
<tr>
<td>IMPLIES</td>
</tr>
<tr>
<td>(rs_valid(q)(y) AND rs_valid(I_step(q,s,i))(y)) IMPLIES</td>
</tr>
<tr>
<td>Complete_instr(q,y) = Complete_instr(I_step(q,s,i),y))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dispatched_remains_induction: CLAIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORALL (rsi:rsindex):</td>
</tr>
<tr>
<td>(rs_valid(q)(rsi) AND Instr_dispatched?(q,rsi) AND NOT Execute?(q,s,rsi)</td>
</tr>
<tr>
<td>AND induction_hypothesis(q,s,i,rsi))</td>
</tr>
<tr>
<td>IMPLIES</td>
</tr>
<tr>
<td>Complete_instr(q,rsi) = Complete_instr(I_step(q,s,i),rsi)</td>
</tr>
</tbody>
</table>

We now sketch the proof of dispatched_remains_induction claim. We expand the Complete_instr function definition on both sides of the consequent. In state \( q, rsi \) is in dispatched phase and in state I_step(q,s,i), it is the dispatched phase too—this follows from the dispatched_remains lemma (not shown, but similar to issued_to_dispatched lemma). After some rewriting and simplifications in PVS, the left hand side of the consequent simplifies to Action_dispatched(q,rsi) and the right hand side simplifies to Action_dispatched(I_step(q,s,i),rsi). The proof now proceeds by expanding the definition of Action_dispatched on both sides of the consequent, using the necessary invariant properties and simplifying using the strategies discussed in Appendix C.

In proving the obligation for the case when an instruction in issued phase remains in that phase, we need to show that the source operand values for the instruction remain the same in the present and the next implementation states. For the first source operand, the relevant definitions and the verification conditions are as shown in 9. The verification condition rs_vall_feedback_VC states that when an instruction obtains its source operand in the current implementation transition (\( rs_ptr1 \) is nonzero and \( rs_ptr1' \) is zero), that value ought to be equal to the value computed by the instruction it is dependent on. Proving this verification condition establishes the correctness of the feedback logic.
In proving `rs_val1_same_VF` verification condition, there are three cases. Consider the case when `rs_ptr1` is zero. We then show that `rs_ptr1` is zero too and `rs_val1` is the same as `rs_val1'`. (The source operand value ought not change when there is no dependency.) Consider the case when `rs_ptr1` is nonzero. `rs_ptr1` may or may not be zero. If `rs_ptr1` is zero, then the proof follows from `rs_val1_feedback_VC`. If `rs_ptr1` is nonzero, then we show that `rs_ptr1` is equal to `rs_ptr1`, and the result then follows from the induction hypothesis.

### 6.2.4.2 Proving the commutativity obligation

We discuss the proof of the commutativity obligation for rf. The proofs for pc and `imem` are straightforward.

We first formulate two verification conditions, one pertaining to an instruction being written back and the other pertaining to a new instruction being issued. These verification conditions are shown in [10]. When an instruction with destination register `r` is being written back ( `rtt` is nonzero and `rtt` is zero), the value in register `r` after an implementation transition ought to be equal to the value computed the the instruction being writtenback. This observation is formulated as verification condition `executed_writeback_same_result_VC`. When a new instruction is issued, the expected result value as given by the specification machine ought to be equal to the value computed by the new instruction (as given by `Complete_instr`) in `I_step(q,s,i)`. This observation is formulated as verification condition `new_instr_rf_same_result_VC`.  

```c
op_val1(q;Reach_state_I,rsi:rsindex): value =
  IF rs_ptr1(q)(rsi) = 0 THEN rs_val1(q)(rsi)
  ELSE Complete_instr(q,rs_ptr1(q)(rsi)) ENDIF
rs_val1_feedback_VC: CLAIM
FORALL(rsi:rsindex):
  (rs_valid(q)(rsi) AND Instr_issued?(q,rsi) AND NOT Dispatch?(q,s,rsi)
   AND rs_ptr1(q)(rsi) /= 0 AND rs_ptr1(I_step(q,s,i))(rsi) = 0)
   IMPLIES
   rs_val1(I_step(q,s,i))(rsi) = Complete_instr(q,rs_ptr1(q)(rsi))
rs_val1_same_VC: CLAIM
FORALL(rsi:rsindex):
  (rs_valid(q)(rsi) AND Instr_issued?(q,rsi) AND NOT Dispatch?(q,s,rsi)
   AND induction_hypothesis(q,s,i,rsi))
   IMPLIES
   op_val1(q,rsi) = op_val1(I_step(q,s,i),rsi)
```
executed_writeback_same_result_VC: CLAIM
FORALL(r:reg_addr):
  (rtt(q)(r) /= 0 AND rtt(I_step(q,s,i))(r) = 0)
  IMPLIES
  select rf(I_step(q,s,i)),r) = Complete_instr(q, rtt(q)(r))

new_instr_rf_same_result_VC: CLAIM
  (NOT no_op?(i) AND NOT rs_valid(q)(sch_rs_slot(s)))
  IMPLIES
  spec_result(projection(Complete_all(q))) =
  Complete_instr(I_step(q,s,i),sch_rs_slot(s))

Now consider the proof of the commutativity obligation when no new instruction is
issued in the current implementation transition (i.e., the synchronization function returns
zero). The commutativity obligation in this case is shown in 11.

rf_commutes_no_issue: LEMMA
  (no_op?(i) OR rs_valid(q)(sch_rs_slot(s)))
  IMPLIES
  rf(abs(I_step(q,s,i))) = rf(A_step_new(abs(q),0))

There are three possible cases in the proof of rf_commutes_no_issue depending on
the register translation table entry for a given register r (refer to the definition of abs
in 5). Consider the case when rtt is zero. We then show that rtt' is zero too and
the values of the register r match in q and I_step(q,s,i). (The register value ought
not change when it is not supposed to be written by any instruction.) Consider the case
when rtt is nonzero. rtt' may or may not be zero. If rtt' is zero, then the result follows
from executed_writeback_same_result_VC. If rtt' is nonzero, then we show that rtt'
is equal to rtt, and the result then follows from same_result_VC.

The proof of the commutativity obligation for 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. Then rtt' is equal to sch_rs_slot(s), and on the specification side of the
commutativity obligation, the value of register r is given by the specification machine.
Then the result follows from new_instr_rf_same_result_VC.

6.2.4.3 Invariant properties needed

All the invariant properties needed in our proof are described below.

- Two of the invariant properties are related to rs_instr_num and instr_counter,
the auxiliary components introduced for defining a measure for every instruction.

  The first property states that the measure of any instruction (rs_instr_num) is
less than the running counter (\texttt{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 operands is less than the measure of the instruction.

- Two other invariant properties are related to \texttt{rs\_exec\_ptr}, the auxiliary component that maintains the execution unit index an instruction is dispatched to. The first property states that, if \texttt{rs\_exec\_ptr} is nonzero, then that execution unit is busy and its tag (which records the instruction executing in the unit) points to the instruction itself. The second property states that, whenever an execution unit is busy, the instruction pointed to by its tag is valid and that instruction’s \texttt{rs\_exec\_ptr} points to the execution unit itself. These properties are similar to the “Instruction phase properties” discussed about \texttt{EX2.2} in Chapter 5.

- Two other properties characterize when an instruction is valid. The first one states that for any register, the instruction pointed to by its register translation table entry (when nonzero) is valid. The second one states that for any given instruction, the instructions pointed to by \texttt{rs\_ptr1} and \texttt{rs\_ptr2} (when nonzero) are valid. The final invariant property needed is about an implementation specific aspect of the machine.
CHAPTER 7

CONCLUSIONS

The earlier chapters illustrated the application of our methodology to verify six example processors exhibiting a wide variety of processor implementation issues. All these results are summarized in Table 7.1, which shows the manual effort spent on each of these examples, listing them in the order verified. The first entry includes the time to learn PVS, the tool in which our methodology is implemented. The first entry in Table 7.1 is an estimate, the next three entries are measured accurate to a day, and the last entry is measured accurate to an hour. A previously reported effort [45] attempting to verify an example comparable in complexity to EX2.2 took 15 person months. Each verification effort in our approach built on the earlier efforts, and reused some of the ideas and the proof machinery. All the PVS specifications and the proof scripts are available at [26].

A major hurdle in the automation of the verification process described in this dissertation is the need to discover suitably strong invariant properties. While the need for some of the invariant properties is obvious in our approach (like the exclusiveness and exhaustiveness of instruction phases), the discovery of many other invariant properties is a manual, time consuming process. Combining the recent advances in the automatic discovery of invariant properties ( [4,52]) to discover some of the invariant properties needed in our approach (especially those pertaining to the control logic) with our method of proof decomposition has the potential to significantly enhance the automation provided

Table 7.1. Examples verified and the effort needed.

<table>
<thead>
<tr>
<th>Example verified</th>
<th>Effort spent doing the proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX1.1 and EX1.2</td>
<td>2 person months</td>
</tr>
<tr>
<td>EX3.1</td>
<td>13 person days</td>
</tr>
<tr>
<td>EX2.1</td>
<td>19 person days</td>
</tr>
<tr>
<td>EX3.2</td>
<td>7 person days</td>
</tr>
<tr>
<td>EX2.2</td>
<td>265 person hours</td>
</tr>
</tbody>
</table>
by our methodology. Another promising aspect is the use of a model checker to discharge some of the verification conditions, which potentially can avoid the need for some invariant properties.

The work presented in this dissertation can be extended in a number of other ways, some of them being:

- Building a system using (a part of) PVS or some other tool that supports decision procedures and rewriting to support the methodology presented in this dissertation. Besides automating parts of the methodology, this system would help the user interactively apply the rest of the process.

- Enhancing our approach further to handle features such as speculative loads, predication, and interrupts of various priorities.

- Enhancing our approach to allow for the verification of other interesting properties such as liveness properties. Our initial experiments indicate reuse of the proof infrastructure presented in this dissertation.

Following our work, two other research groups are using similar ideas. Berezin, Biere, Clarke and Zhu [6] report on using “a light-weight completion function” in a model checking framework incorporating uninterpreted functions for the verification of a processor implementing Tomasulo’s algorithm without a reorder buffer. Cook, Launchbury, Matthews and Kieburtz [16] use an approach to verify explicitly parallel microprocessor designs where the “transformers” they use is similar to the completion functions proposed by us.

In summary, this dissertation demonstrates that the completion functions approach can be used effectively to formally verify the functional correctness of a wide range of pipelined processor designs at the micro-architectural level of abstraction. None of the previous works have been demonstrated on a wide variety of examples uniformly as ours. The key step in the verification is to define a suitable abstraction function while proving the commutativity obligation. Our approach articulates a systematic procedure by which a designer can formulate a very intuitive set of completion functions that help define a suitable abstraction function. The dissertation illustrates how such a construction of the abstraction function leads to a very natural decomposition of the proof of the commutativity obligation. It presents a systematic way to generate the different verification
conditions in the decomposition. It is illustrated how these verification conditions can then be discharged with a high degree of automation using techniques based on decision procedures and rewriting. The verification proceeds incrementally making debugging and error tracing easier. Design iterations are also greatly facilitated by our approach due to the incremental nature of the verification, as changes to a pipeline stage do not cause ripple-effects of changes across the whole verification effort; global re-verification can be avoided because of the layered nature of the verification process. While primarily applicable for in-order completion machines, it is shown that the approach can also be applied to out-of-order completion machines with minor modifications. Our methodology is illustrated on six example processors exhibiting a wide variety of implementation issues with reasonable manual effort.

How does the work presented in this dissertation affect the broader problem of microprocessor validation in industry? As pointed out by Bose, Conte and Austin in an article surveying the state-of-the-art in microprocessor validation [7], several levels of abstraction in microprocessor modeling will be necessary to handle the complexity of the current designs, and techniques to ensure interlevel model integrity are required. The completion functions approach, with the enhancements described before, is a promising solution to bridge the functional correctness gap between the “pre-RTL, cycle accurate functional model” level [7] and the Instruction Set Architecture model level.
APPENDIX A

BRIEF INTRODUCTION TO PVS

This appendix briefly describes the tool used to implement our methodology, namely PVS. PVS provides an integrated environment for the development and analysis of formal specifications, and supports a wide range of activities involved in creating, analyzing and managing theories and proofs. Only the relevant aspects of the tool are explained below. For more details about the tool, refer to the PVS System Guide, the PVS Prover Guide and the PVS Language Reference available at http://pvs.csl.sri.com/.

A.1 The PVS Language

The specification language of PVS is built on higher-order logic, and it provides a rich set of built-in constructs for expressing a variety of notions. The PVS specification (not to be confused with the “Instruction Set Architecture” specification) is organized as a collection of theories. A theory is essentially made up of declarations, which are used to introduce types, constants, variables, formulas etc. A theory, for instance, may contain the definitions of the implementation transition function and the other relevant signal definitions (constant declarations), or the formulation of the different verification conditions and the commutativity obligation (formula declarations). A theory may import certain other theories. For example, the theory containing the formulation of the different verification conditions imports both the theory containing the implementation machine definitions and the theory containing the specification machine definitions. The theory organization in all the examples described in this dissertation is shown in Appendix D.

Type declarations: Type declarations are used to introduce new type names. These form the simplest of type expressions and can be used to construct more complex type expressions. For example,

```plaintext
reg_addr: TYPE+
```

introduces `reg_addr` as an uninterpreted non-empty type. This is used to model the set of register identifiers in all our implementation models. The use of uninterpreted types
helps in providing the right level of abstraction in a PVS specification without restricting
the possible implementations unnecessarily. PVS allows several other type declarations
such as interpreted type declarations, enumerated type declarations etc. Enumerated type
declaration, for instance, is useful in describing the allowed opcodes in an implementation.
For example, in the EX1.1 model considered in Chapter 4, the supported opcodes could
be introduced as:

\[
\text{opcode: \ TYPE* = \{jmp, beqz, load, store, alu_immed, alu_reg\}.}
\]

More complex types can be constructed by using type constructors. Some of the type
constructors available are for constructing functions types, record types, tuple types etc.
Given the above model of register identifiers, the register file can be modeled as a function
returning the value stored in a register, given its identifier. This can be modeled in PVS
by introducing a function type as:

\[
[\text{reg_addr} \rightarrow \text{value}]
\]

and declaring the register file to be of this type. The register file will then be a function
with domain of type reg_addr and range of type value.

Record types are of the form:

\[
[# \ a_1:t_1, \ldots, a_n:t_n #]
\]

where the \(a_i\) are the names of the different components of the record and the \(t_i\) are type
expressions. If \(\text{rec}\) is of the above type, then the component \(a_i\) of \(\text{rec}\) can be accessed
as \(a_i(\text{rec})\) and this will be of type \(t_i\). Record type is useful for modeling, for instance,
Tuple types are similar except that the components have no names, are ordered, and
accessed using proj_1, proj_2 etc.

Constant declarations: Constant declarations are used to introduce new constants, and
in PVS, the term constant refers to functions and relations, as well as the usual (0-ary)
constants. The declaration:

\[
\text{RB: posnat}
\]

introduces \(\text{RB}\) as an uninterpreted positive natural constant. The size of the reorder buffer
in example EX2.2 is modeled this way. The constant interpreted function declarations
are of the form:

\[
\text{fname(arg}_1:t_1, \ldots, \text{arg}_n:t_n): \ rtype = \text{A PVS Expression}
\]

where \(\text{fname}\) is the name of the function, \(\text{arg}_i\) are the names of the arguments, \(t_i\) are
type expressions, and \(rtype\) is the type of the return element. PVS allows for a variety
of expression constructs such as logical and arithmetic operators, function applications, lambda abstractions, if-then-else expressions, record and tuple expressions etc. Constant declarations are used to introduce the various signal definitions, the transition function definition, etc. in all our implementation models.

**Formula declarations:** Formula declarations are used to introduce axioms or theorems. The axioms are introduced using the keyword Axiom, are assumed to be TRUE, and are useful in partially specifying the behavior of some aspect of the implementation machine (such as the scheduler in Ex2.2). The theorems can be introduced using many equivalent keywords such as CLAIM, LEMMA, or THEOREM. These correspond to properties we want to prove about the implementation machine such as the verification conditions etc. In all our examples, the keyword THEOREM is used for the four obligations shown in 3 in Chapter 2, the keyword CLAIM for the verification conditions, and the keyword LEMMA for the other properties and simple obligations (such as the various lemmas used as rewrite rules).

### A.2 The PVS Prover

The PVS Prover provides a variety of commands to construct the proofs of the different theorems.

**PVS Proof display:** Typically, the PVS Prover is used interactively to construct the proof of a theorem and it uses the sequent-style proof representation to display the current proof goal for the proof in progress. The prover maintains a proof tree for the current theorem being proved, and it the aim of the user to construct a proof tree that is complete, in the sense that all the leaves are recognized as TRUE. Each node of the proof tree is a proof goal that results from the application of a proof step to its parent node. Each proof goal is a sequent consisting of two sequences of formulas called the antecedents and the consequents, displayed by PVS as shown in 1

```
| [-1] A1
| [-2] A2
| [-3] A3
| ...
|---------------------
| [1] C1
| [3] C3
| ...
```
The \( A_1 \) are the antecedent formulas and the \( C_1 \) are the consequent formulas; and the intuitive interpretation of the sequent is that the conjunction of the antecedents formulas implies the disjunction of the consequent formulas. When the prover is invoked on a theorem to be proved, the proof tree starts off with a root node having no antecedent formulas and the theorem to be proved as the only one consequent formula. At any time in a PVS proof, attention is focused on some sequent-proof goal that is a leaf node in the current proof tree, and PVS displays this sequent while awaiting the user's command. Invoking a prover command may cause the current proof goal to be recognized as \texttt{TRUE} or may generate more subtrees. Once a sequent is recognized to be \texttt{TRUE}, that branch of the proof tree is terminated. The aim is to build a proof tree whose branches have all been terminated this way.

\textit{PVS Prover commands:} Some of the prover commands that are commonly used and a brief explanation about it (many of these commands take arguments that control its behavior which are not discussed here):

- **expand:** This command expands and simplifies the definitions of the specified functions/predicates at the specified occurrences.

- **lemma:** This command is used to pull in a previously proved theorem into the current proof goal instantiated as specified by the user.

- **auto-rewrite!** This command instructs the prover to use the specified theorems and definitions as rewrite rules in the subsequent proof steps.

- **auto-rewrite-theory:** This command installs all the definitions and the theorems in the specified theories as rewrite rules to be used by the subsequent proof steps.

- **stop-rewrite:** This command turns off the specified rewrite rules.

- **case:** This command generates two subgoals, one where the given boolean expression is assumed to be \texttt{TRUE} and the other where it is assumed to be \texttt{FALSE}.

- **bddsimp:** This command generates subgoals by applying propositional simplification using an external package written in C language and based on binary decision diagrams [11].
• **flatten**: This command is used to break an antecedent formula that is a conjunction or a consequent formula that is a disjunction into its components (conjuncts/disjuncts).

• **lift-if**: This command is used to lift the “IF branching structure” embedded within a formula to the topmost level, so that propositional simplification using other commands can then be applied.

• **inst**: This command is used to instantiate a universally quantified antecedent formula or an existentially quantified consequent formula.

• **assert**: This command is used to simplify the proof goal using decision procedures and rewriting. The decision procedures employ a congruence closure algorithm for equality reasoning and they also perform linear arithmetic reasoning over natural numbers.

• **ground**: This command invokes propositional simplification followed by simplification using decision procedures and rewriting.

PVS also provides a simple language to combine sequences of commonly used proof steps into strategies. These strategies can then be used as normal prover commands.

The PVS System provides an Emacs interface to interactively construct the proof and a Tcl/Tk interface to graphically display the proof trees. The PVS System also provides other commands to rerun, maintain, and modify the existing proofs.
APPENDIX B

MODELING THE MACHINES

This appendix discusses the modeling of the implementation and the specification machines, and other structures such as the register file and the data memory in our approach. It also discusses some simple techniques to access the various components of the machines during a proof in PVS.

B.1 Modeling the Different Components in the Machines

A processor computes “values” while executing an instruction such as the final result of the instruction to be stored in a register or in a memory location, or a memory address or an instruction address etc., and an uninterpreted type value is introduced to model them. Some implementation models use certain constants of type value such as zero, four etc. (their use described below). Some other implementation models use the uninterpreted predicate zero_val? that tests whether a given value is zero or not.\(^1\)

The register file is modeled as a function that returns the value of a register, given its identifier. An uninterpreted type reg_addr is used to model the register identifiers. Two operations select and assign are defined with their obvious meanings: select for reading a given register and assign for writing a given value to a given register. Two properties are then proved about these operations. Assume a value d1 is assigned to a register r1. The first property states that reading register r1 returns d1. The second property states that as far as reading a different register is concerned, the original assignment is irrelevant. The definitions of these operations and two properties about them are shown in [1]. Some implementation models considered in this dissertation have a fixed register named zero_reg. Reading this register always returns zero and writing

\(^1\)An implementation model that uses the constant zero does not use the uninterpreted predicate zero_val?.
this register does not modify it.

```
reg_addr: TYPE
regfile: TYPE = [reg_addr -> value]

select(rf:regfile,r:reg_addr):value = rf(r)
assign(rf:regfile,r:reg_addr,data:value):regfile = rf WITH [(r) := data]

select_assign1: LEMMA
  FORALL((r1,r2:reg_addr),rf:regfile,d1:value):
    r2 = r1 IMPLIES select(assign(rf,r1,d1),r2) = d1

select_assign2: LEMMA
  FORALL((r1,r2:reg_addr),rf:regfile,d1:value):
    r2 /= r1 IMPLIES select(assign(rf,r1,d1),r2) = select(rf,r2)
```

The data memory is modeled in a similar way as the register file, with two operations read and write, and two exactly similar properties. The data memory address is of type dmem_addr. The memory addresses are normally computed by the processor during the execution of an instruction, and an uninterpreted function d_addr—that takes two arguments of type value and returns a value—is used for this purpose. Another function value2dmem_addr is then used to convert it to dmem_addr type, and this function is partially interpreted by means of an axiom that states that distinct values map to distinct data memory addresses.

The instructions are modeled by an uninterpreted type instruction and the supported opcodes by an enumerated type opcode. Four uninterpreted functions op, src1, src2 and dest extract the opcode, the first source register identifier, the second source register identifier, and the destination register identifier from an instruction word respectively. Some more such functions (for example, for extracting immediate fields from the instruction) are used in some implementation models. The instruction memory is modeled as a function from instruction addresses to instructions. imem_addr, value2imem_addr and i_add have similar meanings as in the case of the data memory. The program counter keeps track of the address of the next instruction to be executed and has type value. The uninterpreted function i_add is used in computing the new instruction address while calculating the branch instruction outcome or in incrementing the program counter. Some implementation models use constant values such as four, eight, etc. in determining the new instruction address, and the necessary partial interpretation is provided on these values using axioms. Some implementation models use an uninterpreted function inc_pc to increment the program counter during an instruction fetch.
We concentrate only on the flow of instructions in the pipeline in our work, abstracting away from details such as the implementation of the arithmetic logic unit or that of the scheduler. The arithmetic logic unit is modeled as an uninterpreted function \texttt{alu} in most implementation models in our work. The scheduler that controls the flow of instructions in the pipeline is modeled by using axioms which capture its allowable behaviors. Similarly, some other details in the implementation machine are left underspecified, when those details are "uninteresting" in our work.

### B.1.1 Instructions supported

The following instructions are supported in most of the implementation models considered in this dissertation, and their semantics when they raise no exceptions are described below. Certain other instructions supported only in some specific models are explained in the appropriate chapters.

- **jmp**: An unconditional branch instruction. The target address is determined by an immediate value obtained from the instruction word and the program counter.

- **beqz**: Conditional branch instruction. The branch is taken only if the first source register value is zero. The target address is determined by an immediate value obtained from the instruction word and the program counter.

- **load**: Loads the value from a memory location into the destination register. The memory address is determined by the first source register value and an immediate value from the instruction word.

- **store**: Stores the value of the second source register in a memory location whose address is determined as in the case of the load instruction.

- Arithmetic and logical instructions: \texttt{alu\_reg} instruction (or \texttt{alu} instruction in some models) performs arithmetic/logical operations on the first and the second source register values, and stores the result in the destination register. \texttt{alu\_immed} is a variant of this instruction in which the second operand is an immediate value obtained from the instruction word.
B.2 Machine Definitions and Accessing the Different Components

As pointed out in Chapter 2, the implementation machine transition function \texttt{I\_step} is specified by describing how each component obtains its new value. For each implementation component \texttt{ic}, a function \texttt{next\_ic} is defined that describes how its new value is obtained. The template of the definition of \texttt{I\_step} is as shown in \(2\). (In models with a scheduler, \texttt{I\_step} takes the scheduler output as another argument.) The specification machine transition function \texttt{A\_step} is similarly specified. These machine component definitions and other intermediate definitions used are sometimes also referred to as signals.

\begin{verbatim}
I_step(q:state_I,i:inputs_type): state_I = ( #
observable_1 := next_observable_1(q,i),
    ....
observable_k := next_observable_k(q,i),
nonobservable_1 := next_nonobservable_1(q,i),
    ....
nonobservable_n := next_nonobservable_n(q,i) #)
\end{verbatim}

During the proofs in PVS, it will be necessary to access the different machine components after a machine transition. This is accomplished by defining "record accessor" lemmas for every machine component, and using these lemmas as automatic rewrite rules in the PVS proofs. For an implementation machine component \texttt{ic}, the record accessor lemma has the form shown in \(3\) and there are similar lemmas for every specification machine component too. The proofs of these lemmas are all trivial.

\begin{verbatim}
ic: LEMMA
  ic(I_step(q,i)) = next_ic(q,i)
\end{verbatim}

In some implementation models, the \texttt{next\_ic} definition for some of the implementation components reduce to a simpler expression under certain conditions. Typically the proofs of the obligations due to certain invariant properties are split into different cases based on these conditions. It is convenient in such proofs to define a lemma capturing this property and use it as an automatic rewrite rule, instead of expanding the definition of \texttt{next\_ic} during the proofs and simplifying it in place. As an example, one such lemma used in Ex2.2 implementation discussed in Chapter 5 is shown in \(4\). It states that when \texttt{Issued at rb\_head} predicate is \texttt{FALSE}, the value of the component \texttt{rb\_dest} for instruction \texttt{rbi} is unchanged after an implementation transition. There are similar
lemmas for some other implementation components too. The proofs of these lemmas are all straightforward, involving expanding the definitions occurring in them and simplifying the resulting expression.

```
new_rewrite_next_rb_dest: LEMMA
  NOT Issued_at_rb_head(q,s,i,rbi)
  IMPLIES
  next_rb_dest(q,s,i)(rbi) = rb_dest(q)(rbi)
```
APPENDIX C

PVS STRATEGIES USED IN OUR APPROACH

This appendix describes some of the commonly used PVS proof strategies.

C.1 Strategies to Install Rewrite Rules

These strategies pertain to installing the machine definitions, the abstraction function related definitions, and certain properties and lemmas proved about them as automatic rewrite rules to be used by the PVS prover for simplification during the proofs. These strategies, when invoked, instructs the prover to use the specified definitions and lemmas as automatic rewrite rules in the subsequent steps in the proofs.

Three strategies are defined regarding the use of implementation machine definitions and properties as rewrite rules. The first strategy Impl-rewrites installs the record accessor and other lemmas about the implementation machine discussed in Appendix B as rewrite rules. The second strategy Impl-props installs certain properties about the implementation machine (that are in a suitable form) as rewrite rules. The third strategy Impl-defs installs most of the implementation machine definitions such as the next_ic definitions of the various implementation components and other signal definitions as rewrite rules. However some definitions are not turned into automatic rewrite rules for reasons described below:

- They occur frequently in many of the signal definitions in the machine, and when expanded at all occurrences in an expression, it leads to a very large expression size. It is desirable to have more control on when to expand such definitions.

- Some signal definitions are used as preconditions in some verification conditions and other properties proved about the implementation machine. Expanding these too early in a proof will not trigger the rewrite rules corresponding to these verification
conditions and properties. It is again desirable to have more control on when to expand such definitions.

- Certain signal definitions such as those defining the source register values after an implementation transition have verification conditions defined regarding them. In most proofs, it is appropriate to use these verification conditions about them instead of their definitions. Hence their definitions are not turned into rewrite rules.

The guidelines mentioned above are used in deciding the definitions to be used as rewrite rules. The decision of whether or not to use a given definition as a rewrite rule may turn out to be an inappropriate one in certain proofs. The situation can then be remedied by using the PVS commands **stop-rewrite** or **auto-rewrite** as necessary (see Appendix A).

Three strategies are defined regarding the use abstraction function related definitions and properties as rewrite rules. The strategy **Abs-rewrites** installs the lemmas that state that the completion functions do not modify the nonobservable components as rewrite rules. (see Section 4.1.4 and Section 5.3.3.1). The strategy **Abs-props** installs some of the abstraction function related properties as rewrite rules. The strategy **Abs-defs** installs most of the definitions related to the abstraction function as rewrite rules. We follow the same guidelines discussed before in deciding to exclude some definitions from being installed as rewrite rules.

Two strategies are defined regarding the use of specification machine definitions as rewrite rules. These strategies are **Spec-rewrites** and **Spec-defs** and are similar to their counterparts for the implementation machine. Finally, the strategy **Store-rewrites** installs the properties about the operations for the register file and the data memory as rewrite rules.

### C.2 Strategies to Simplify Expressions

The main simplification strategy used is a brute-force case analysis strategy that attempts to simplify the proof goal to **true** under the different possible cases. This strategy called **case-analysis** is shown in [1]. It proceeds by repeatedly lifting all the **IF-THEN-ELSE** conditionals to the top-most level, simplifying the resulting proof goal using a BDD [11] based simplifier, followed by further simplification using rewriting and decision procedures.
(defstep case-analysis ()
  (apply (then (repeat (lift-if)) (bddsimp) (ground) (assert)))
  "Brute force simplification...."
  "Doing case analysis...")

Few other commonly used sequences of proof commands are turned into strategies too. Some of them are discussed below:

- In many proofs, it is necessary to break the antecedent formulas that appear as a conjunction into its conjuncts, and the consequent formulas that appear as a disjunction into its disjuncts, and the strategy fa* shown in 2 accomplishes this.

- While the case analysis strategy discussed earlier is quite effective in practice, sometimes the sequence in which it does the case analysis might not be appropriate and may take a long time. We can then guide the case analysis by explicitly suggesting a case-split and simplifying the two branches in the respective contexts (the strategy ca shown in 2 does this). It is then followed by a brute-force case analysis on both the branches.

(defformula fa* ()
  (repeat* (then (flatten) (assert)))
  "Repeatedly flatten and assert"
  "Flattening and asserting repeatedly.."
)

(defstep ca (expr)
  (spread (case expr)
    ((apply (then (replace -1) (assert) (fa*)))
     (apply (then (replace 1) (assert) (fa*)))))
  "Do a case split. Replace in both branches and do an assert, then fa*
  "Do a case split. Replace in both branches and do an assert, then fa*")
APPENDIX D

PVS PROOF ORGANIZATION AND AN EXAMPLE PROOF

This appendix describes the PVS theory organization in all the examples described in this dissertation. The second section explains the proof of one obligation in detail.

D.1 Theory Organization

Figure D.1 shows the organization of the PVS theories in all the examples described in this dissertation. A brief description of each of the theories follows:

- **stores**: This theory contains the definitions of the register file, the data memory, and the operations defined on them.

- **instructions**: This theory contains the definitions of the instructions supported and certain other relevant definitions.

- **choose_theory**: This theory defines a variant of the “choice” operator of PVS used in the definition of the implementation transition function.

- **specification**: This theory contains the ISA level specification of the machine.

- **specification_rewrites**: This theory contains the “record accessor” lemmas about the specification machine that are used as rewrite rules.

- **implementation_state**: This theory contains the definitions of the implementation machine state and the invariant properties.

- **implementation_trans**: This theory contains the implementation transition function definition.

- **implementation_rewrites**: This theory contains the lemmas about the implementation machine discussed in Appendix B that are used as rewrite rules.
Figure D.1. Theory organization in all the examples described in this dissertation
- **implementation.props**: This theory contains certain simple properties about the implementation machine.

- **implementation.invariants**: This theory contains the lemmas that lead up to the proof of the Inv_Init and Inv_Closure obligations (see [3] in Chapter 2).

- **abstraction**: This theory contains the definitions of the completion functions, the squashing predicates and the abstraction function.

- **abstraction_rewrites**: This theory contains certain lemmas about the completion functions that are used as rewrite rules (see Section 4.1.4 and Section 5.3.3.1).

- **abstraction.props**: This theory contains certain simple properties related to the abstraction function.

- **commutativity**: This theory contains the formulation of the different verification conditions and the commutativity obligation.

- **projection**: This theory contains the definition of the Flushed? predicate and the formulation of the projection obligation.

### D.2 An Example Proof

This section explains the proof of the abs_issued_remains_induction obligation for the observable rf in detail (elaborating further on the description in Section 5.3.3.2). This case of the obligation is shown in [1].

```
abs_issued_remains_induction_rf: LEMMA
   FORALL(rbi_ms:rbindex):
   LET rbi = measure_fn_rbi(q,rbi_ms) IN
   ((valid_rb_entry?(q,rbi) AND NOT restart_proc?(q,s)
      AND Instr_issued_abs?(q,rbi) AND NOT Dispatch_abs?(q,s,rbi)
      AND rb_induction_hypothesis(q,s,i,rbi_ms))
   IMPLIES
   rf(Complete_till(q,rbi_ms)) = rf(Complete_till(i_step(q,s,i),
      rbi_measure_adjust(q,s,rbi_ms))))
```

Invoking the PVS prover on this obligation pops up the prover with the proof goal shown in [2].
The universally quantified variables are skolemized, a name rbi!1 is introduced for measure_fn_rbi(q!1,rbi_ms!1), and the proof goal is simplified using the strategy fa*.

Pulling in the lemma abs issued remains into the proof and simplifying further leads to the proof goal shown in 3.

We now want to consider the effect of completing the instruction rbi!1 on rf. So the Complete_till function definition is expanded, and the Complete_Squash_rest?_till function definition (that appears when Complete_till is expanded) is then expanded. Some of the earlier proved verification conditions and some of the properties about the implementation machine (when they are in a suitable form) are installed as automatic rewrite rules. Also the abstraction function related rewrite rules discussed in Appendix C are installed as rewrite rules by invoking the strategies defined for that purpose (Abs-defs, Abs-rewrites and Abs-props). The proof goal is simplified using these rewrite rules and decision procedures (by invoking the strategy fa*), and the proof goal reduces to one shown in 4.
At this point, some of the properties about \( \text{rbi!1} \) are needed in the proof. Since \( \text{rbi!1} \) is in \textit{issued.abs} phase in both states \( q!1 \) and \( \text{I.step}(q!1,s!1,i!1) \), it can only be in that phase in these two states by the \textit{Instruction exclusive} invariant property. To use this fact, the \textit{Instruction exclusive} invariant property (instantiated in both states \( q!1 \) and \( \text{I.step}(q!1,s!1,i!1) \)) is pulled into the proof. Also the relevant "instruction phase property" discussed in Section 5.3.2.2 ( \texttt{Rb_pointer.abs.rs_valid} shown in in Chapter 5 in this case) is pulled into the proof. The instruction \( \text{rbi!1} \) has the same source operand values before and after the implementation transition, and it is necessary to use this fact in the proof. So the verification conditions \texttt{abs.src1.value_same.VC} and \texttt{abs.src2.value_same.VC} (second one shown in in Section 5.3.1.3) that relate the source operand values before and after an implementation transition are pulled into the proof. The \texttt{Complete.till} and \texttt{Squash_rest?_till} function definitions occurring in these verification conditions are expanded, but the definitions \texttt{abs.src1.value} and \texttt{abs.src2.value} are not expanded (the reason will become apparent later). After simplification, the proof goal reduces to one shown in \[ 5 \].
{-1} (NOT PROJ_2(Complete_Squash_rest? till(Complete_committed_in_sb_till
 (q'1, lsmu sb_commit_count(q'1)), rbi_ms!1 - 1))
 IMPLIES
 abs_src2_value(PROJ_1(Complete_Squash_rest? till(Complete_committed_in_sb_till
 (q'1, lsmu sb_commit_count(q'1)), rbi_ms!1 - 1)),
 measure_fn_rbi(q'1, rbi_ms!1))
 =
 abs_src2_value(PROJ_1(Complete_Squash_rest? till(Complete_committed_in_sb_till
 (I_step(q'1, s'1, i'1)), lsmu sb_commit_count(I_step(q'1, s'1, i'1))),
 rbi_measure_adj(q'1, s'1, rbi_ms!1) - 1)),
 measure_fn_rbi(q'1, rbi_ms!1)))

{-2} 'Similar formula regarding abs_src1_value'
{-3} abs_src1_valid(q'1)(rb_abs_rs(q'1)(rb!1))
{-4} abs_src1_dest(q'1)(rb_abs_rs(q'1)(rb!1)) = rb!1
{-5} rb_abs_eu(I_step(q'1, s'1, i'1))(rb!1) = 0
{-6} rb_lsmu(I_step(q'1, s'1, i'1))(rb!1) = 0
{-7} rb_abs_eu(q'1)(rb!1) = 0
{-8} rb_lsmu(q'1)(rb!1) = 0
{-9} rb_abs_eu(I_step(q'1, s'1, i'1))(measure_fn_rbi(q'1, rbi_ms!1)) > 0
{-10} valid_rb_entry?(I_step(q'1, s'1, i'1), measure_fn_rbi(q'1, rbi_ms!1))
{-11} rb_abs_eu(I_step(q'1, s'1, i'1))(measure_fn_rbi(q'1, rbi_ms!1)) =
 rb_abs_eu(q'1)(measure_fn_rbi(q'1, rbi_ms!1))
{-12} measure_fn_rbi(q'1, rbi_ms!1) = rb!1
{-13} valid_rb_entry?(q'1, rbi!1)
{-14} Instr_issued_abs?(q'1, rbi!1)
{-15} rb_induction_hypothesis(q'1, s'1, i'1, rbi_ms!1)

[1] rb_lsmu_eu(I_step(q'1, s'1, i'1))(rb!1)
[2] rb_ready(I_step(q'1, s'1, i'1))(rb!1)
[3] rb_abs_eu(I_step(q'1, s'1, i'1))(rb!1) > 0
[4] rb_lsmu(I_step(q'1, s'1, i'1))(rb!1) > 0
[5] rb_lsmu_eu(I_step(q'1, s'1, i'1))(rb!1)
[6] rb_ready(I_step(q'1, s'1, i'1))(rb!1)
[7] rb_lsmu_eu(q'1)(rb!1)
[8] rb_ready(q'1)(rb!1)
[9] rb_abs_eu(q'1)(rb!1) > 0
[10] rb_lsmu_eu(q'1)(rb!1) > 0
[12] rb_ready(q'1)(rb!1)
[13] rbi_measure_adj(q'1, s'1, rbi_ms!1) = 0
[14] restart_proc?(q'1, s'1)
 sch_abi_di(s'1)(abs_eui) = rb_abs_rs(q'1)(rb!1))
[16] rf(Complete_instr(PROJ_1(Complete_Squash_rest? till(Complete_committed_in_sb_till
 (q'1, lsmu sb_commit_count(q'1)), rbi_ms!1 - 1)),
 measure_fn_rbi(q'1, rbi_ms!1),
 PROJ_2(Complete_Squash_rest? till(Complete_committed_in_sb_till
 (q'1, lsmu sb_commit_count(q'1)), rbi_ms!1 - 1)))
 =
 rf(Complete_instr(PROJ_1(Complete_Squash_rest? till(Complete_committed_in_sb_till
 (I_step(q'1, s'1, i'1)), lsmu sb_commit_count(I_step(q'1, s'1, i'1))),
 rbi_measure_adj(q'1, s'1, rbi_ms!1) - 1)),
 measure_fn_rbi(q'1, rbi_ms!1),
 PROJ_2(Complete_Squash_rest? till(Complete_committed_in_sb_till
 (I_step(q'1, s'1, i'1)), lsmu sb_commit_count(I_step(q'1, s'1, i'1))),
 rbi_measure_adj(q'1, s'1, rbi_ms!1) - 1)))
Now consider the basis case of the induction argument, that is, $\text{rbi_ms}!1 = 1$. In this case, $\text{rbi!}1$ is the tail of the reorder buffer. Since $\text{rbi!}1$ is in $\text{issued_abs}$ phase, the scheduler can not retire it (only an instruction in $\text{writtenback}$ phase can be retired), and hence $\text{sch_retire}\_\text{rb?}(s!1)$ is $\text{FALSE}$. This can be inferred from an axiom constraining the behavior of the scheduler. Referring to the definition $\text{rbi\_measure\_adjust}$ in [10] in Section 5.3.1.2, the adjusted measure remains the same when $\text{sch_retire}\_\text{rb?}(s!1)$ is $\text{FALSE}$. So in the formulas numbered $-1, -2,$ and $16$ in [5], on the right hand side, the second argument to $\text{Complete\_Squash\_rest?\_till}$, that is, $\text{rbi\_measure\_adjust}(q!1,s!1,\text{rbi\_ms}!1) - 1$, simplifies to zero. The second argument to $\text{Complete\_Squash\_rest?\_till}$ is already zero on the left hand side in these formulas. The $\text{Complete\_Squash\_rest?\_till}$ function definitions in these formulas are expanded. The $\text{kill?}$ argument to $\text{Complete\_instr}$ in formula numbered $16$ (in [5]) is now $\text{FALSE}$. The $\text{Complete\_instr}$ function definition on both sides of the formula numbered $16$ are expanded and the proof goal simplified by using rewriting and decision procedures (using the strategy $\text{fas}$). Since the instruction $\text{rbi!}1$ is in $\text{issued\_abs}$ phase in state $q!1$, it is in $\text{issued\_abs}$ phase in the state $\text{Complete\_committed\_in\_sb\_till}(q!1,\text{lsu\_sb\_commit\_count}(q!1))$ too. This follows from the fact that the completion functions do not modify the nonobservable components and the various phase definitions refer to the nonobservable components only. So on the left hand side of formula numbered $16$, the case corresponding to the $\text{issued\_abs}$ phase applies (see the definition of $\text{Complete\_instr}$ in [2] in Section 5.2.2) and a similar observation holds regarding the right hand side. The proof goal simplifies one shown in [6].

Proof now proceeds by expanding the various function definitions occurring in the proof goal and then attempting to simplify the proof goal to $\text{TRUE}$. In a correct implementation, it ought to simplify to $\text{TRUE}$. First $\text{Action\_issued\_abs}$ function definition is expanded on both sides of the formula numbered $17$ in [6]. A fragment of this formula after simplification is shown in [7].

Observe that the terms corresponding to the source operand values on both sides are equal by formulas $-2$ and $-3$ in [6]. Also, $\text{rb\_abs\_rs'}$ is equal to $\text{rb\_abs\_rs}$ by formula $-12$. So the fragment shown in [7] further simplifies one shown in [8].

Now, all the implementation machine terms occurring in the proof goal are expanded (we hide certain irrelevant formulas), two properties about the implementation machine
are pulled in, and the resulting expression is simplified by brute force case analysis. It ought to simplify to TRUE in a correct implementation.

```plaintext
[-1] rbi_ms!1 = 1
[-2] abs_src2_value(Complete_commited_in_sb_till(q!1,
   lsu_sb_commit_count(q!1)), rb_tail(q!1)) =
   abs_src2_value(Complete_commited_in_sb_till(l_step(q!1, s!1, i!1),
   lsu_sb_commit_count(l_step(q!1, s!1, i!1))), rb_tail(q!1))
[-3] abs_src1_value(Complete_commited_in_sb_till(q!1,
   lsu_sb_commit_count(q!1)), rb_tail(q!1)) =
   abs_src1_value(Complete_commited_in_sb_till(l_step(q!1, s!1, i!1),
   lsu_sb_commit_count(l_step(q!1, s!1, i!1))), rb_tail(q!1))
[-4] abs_src_valid(q!1)(rb_abs_rs(q!1)(rbi!1))
[-5] abs_src_dest(q!1)(rb_abs_rs(q!1)(rbi!1)) = rbi!1
[-6] rb_abs_eu(l_step(q!1, s!1, i!1))(rbi!1) = 0
[-7] rb_lsu_rs(l_step(q!1, s!1, i!1))(rbi!1) = 0
[-8] rb_abs_eu(q!1)(rbi!1) = 0
[-9] rb_lsu_rs(q!1)(rbi!1) = 0
[-10] rb_abs_rs(l_step(q!1, s!1, i!1))(rb_tail(q!1)) > 0
[-11] valid_rb_entry?(l_step(q!1, s!1, i!1), rb_tail(q!1))
[-12] rb_abs_rs(l_step(q!1, s!1, i!1))(rb_tail(q!1)) =
   rb_abs_rs(q!1)(rb_tail(q!1))
[-13] rb_tail(q!1) = rbi!1
[-14] valid_rb_entry?(q!1, rbi!1)
[-15] Instrissued_abs?(q!1, rbi!1)
[-16] rb_induction_hypothesis(q!1, s!1, i!1, 1)

----------
[1] sch_retire rb?(s!1)
[2] rb_lsu_eu(l_step(q!1, s!1, i!1))(rbi!1)
[3] rb_ready(l_step(q!1, s!1, i!1))(rbi!1)
[4] rb_abs_eu(l_step(q!1, s!1, i!1))(rbi!1) > 0
[5] rb_lsu_rs(l_step(q!1, s!1, i!1))(rbi!1) > 0
[6] rb_lsu_eu(l_step(q!1, s!1, i!1))(rbi!1)
[7] rb_ready(l_step(q!1, s!1, i!1))(rbi!1)
[8] rb_lsu_eu(q!1)(rbi!1)
[9] rb_ready(q!1)(rbi!1)
[10] rb_abs_eu(q!1)(rbi!1) > 0
[11] rb_lsu_rs(q!1)(rbi!1) > 0
[12] rb_lsu_eu(q!1)(rbi!1)
[13] rb_ready(q!1)(rbi!1)
[14] rbi_measure_adjust(q!1, s!1, 1) = 0
[15] restart_proc?(q!1, s!1)
   sch_abs_di(s!1)(abs_eui) = rb_abs_rs(q!1)(rbi!1))
[17] rf(Actionissued_abs(Complete_commited_in_sb_till(q!1,
   lsu_sb_commit_count(q!1)), rb_tail(q!1))) =
   rf(Actionissued_abs(Complete_commited_in_sb_till(l_step(q!1, s!1, i!1),
   lsu_sb_commit_count(l_step(q!1, s!1, i!1))), rb_tail(q!1)))
```
IF NOT abs_rs_op(q!1)(rb_abs_rs(q!1)(rb_tail(q!1))) = ALU_E AND
    rb_reg_write?(q!1)(rb_tail(q!1))
THEN assign(rf(q!1), rb_dest(q!1)(rb_tail(q!1))),
    IF abs_rs_op(q!1)(rb_abs_rs(q!1)(rb_tail(q!1))) = ALU
    THEN alu(abs_rs_op(q!1)(rb_abs_rs(q!1)(rb_tail(q!1)));
        abs_src1_value(Complete_commited_in_sb_till(q!1),
        lsu_sb_committ_count(q!1), rb_tail(q!1)),
        abs_src2_value(Complete_commited_in_sb_till(q!1),
        lsu_sb_committ_count(q!1), rb_tail(q!1))
    ELSIF ‘‘...Truncated...’’  ENDIF
ELSE rf(q!1) ENDIF

ELSE rf(I_step(q!1,s!1,i!1)) ENDIF

IF NOT abs_rs_op(I_step(q!1,s!1,i!1))(rb_abs_rs(q!1)(rb_tail(q!1))) = ALU_E AND
    rb_reg_write?(I_step(q!1,s!1,i!1))(rb_tail(q!1))
THEN assign(rf(I_step(q!1,s!1,i!1)), rb_dest(I_step(q!1,s!1,i!1))(rb_tail(q!1)),
    IF abs_rs_op(I_step(q!1,s!1,i!1))(rb_abs_rs(I_step(q!1,s!1,i!1)))(rb_tail(q!1))) = ALU
    THEN alu(abs_rs_op(I_step(q!1,s!1,i!1))(rb_abs_rs(I_step(q!1,s!1,i!1)))(rb_tail(q!1)));
        abs_src1_value(Complete_commited_in_sb_till(I_step(q!1,s!1,i!1),
        lsu_sb_committ_count(I_step(q!1,s!1,i!1)), rb_tail(q!1)),
        abs_src2_value(Complete_commited_in_sb_till(I_step(q!1,s!1,i!1),
        lsu_sb_committ_count(I_step(q!1,s!1,i!1)), rb_tail(q!1))
    ELSIF ‘‘...Truncated...’’  ENDIF
ELSE rf(I_step(q!1,s!1,i!1)) ENDIF

Now consider the case when rbi ms!1 is not equal to one. In this case, the induction 
hypothesis expands into the expression shown in [9].
valid_rb_entry?(q!1, measure_fn_rbi(q!1, rbi_ms!1 - 1)) IMPLIES 
   (rf(Complete_till(q!1, rbi_ms!1 - 1)) = 
    rf(Complete_till(I_step(q!1, s!1, i!1), rbi_measure_adjust(q!1, s!1, rbi_ms!1 - 1))) 
   AND 
   sp_rf(Complete_till(q!1, rbi_ms!1 - 1)) = 
    sp_rf(Complete_till(I_step(q!1, s!1, i!1), rbi_measure_adjust(q!1, s!1, rbi_ms!1 - 1))) 
   AND 
   mode_reg(Complete_till(q!1, rbi_ms!1 - 1)) = 
    mode_reg(Complete_till(I_step(q!1, s!1, i!1), rbi_measure_adjust(q!1, s!1, rbi_ms!1 - 1))) 
   AND 
   dmem(Complete_till(q!1, rbi_ms!1 - 1)) = 
    dmem(Complete_till(I_step(q!1, s!1, i!1), rbi_measure_adjust(q!1, s!1, rbi_ms!1 - 1))) 
   AND 
   Squash_rest_till(q!1, rbi_ms!1 - 1) = 
    Squash_rest_till(I_step(q!1, s!1, i!1), rbi_measure_adjust(q!1, s!1, rbi_ms!1 - 1)))

Since rbi!1 is a valid instruction, measure_fn_rbi(q!1, rbi_ms!1 - 1) is valid too (follows from an implementation property). Also, in the formulas numbered -1, -2 and 16 (refer to [5]), the expression rbi_measure_adjust(q!1, s!1, rbi_ms!1) - 1 is equal to rbi_measure_adjust(q!1, s!1, rbi_ms!1 - 1). The kill? argument to Complete_instr on both sides in the formula numbered 16 (in [5]) has the same truth value, and this follows from the induction hypothesis (fifth conjunct in [9]). When the kill? argument is TRUE, the proof follows immediately from the induction hypothesis (first conjunct in [9]). Otherwise, the formula numbered 16 can be simplified to the one shown in [10] (for the same reasons as in the basis case).

Proof now proceeds as in the basis case. The values of the observable components rf, sp_rf, mode_reg and dmem on both sides of the formula 16 are equal by the induction hypothesis, and the source operand values are equal by the formulas -1 and -2.
REFERENCES


and Kropf [42], pp. 8–22.


