Formal Verification of Instruction Dependencies in Microprocessors

by

Hazem Shehata

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2011

© Hazem Shehata 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Hazem Shehata

Abstract

In microprocessors, achieving an efficient utilization of the execution units is a key factor in improving performance. However, maintaining an uninterrupted flow of instructions is a challenge due to the data and control dependencies between instructions of a program. Modern microprocessors employ aggressive optimizations trying to keep their execution units busy without violating inter-instruction dependencies. Such complex optimizations may cause subtle implementation flaws that can be hard to detect using conventional simulation-based verification techniques.

Formal verification is known for its ability to discover design flaws that may go undetected using conventional verification techniques. However, with formal verification come two major challenges. First, the correctness of the implementation needs to be defined formally. Second, formal verification is often hard to apply at the scale of realistic implementations.

In this thesis, we present a formal verification strategy to guarantee that a microprocessor implementation preserves both data and control dependencies among instructions. Throughout our strategy, we address the two major challenges associated with formal verification: correctness and scalability.

We address the correctness challenge by specifying our correctness in the context of generic pipelines. Unlike conventional pipeline hazard rules, we make no distinction between the data and control aspects. Instead, we describe the relationship between a producer instruction and a consumer instruction in a way such that both instructions can speculatively read their source operands, speculatively write their results, and go out of their program order during execution. In addition to supporting branch and value prediction, our correctness criteria allow the implementation to discard (squash) or replay instructions while being executed.

We address the scalability challenge in three ways: abstraction, decomposition, and induction. First, we state our inter-instruction dependency correctness criteria in terms of read and write operations without making reference to data values. Consequently, our correctness criteria can be verified for implementations with abstract datapaths. Second,

we decompose our correctness criteria into a set of smaller obligations that are easier to verify. All these obligations can be expressed as properties within the Syntactically-Safe fragment of Linear Temporal Logic (SSLTL). Third, we introduce a technique to verify SSLTL properties by induction, and prove its soundness and completeness.

To demonstrate our overall strategy, we verified a term-level model of an out-of-order speculative processor. The processor model implements register renaming using a P6-style reorder buffer and branch prediction with a hybrid (discard-replay) recovery mechanism. The verification obligations (expressed in SSLTL) are checked using a tool implementing our inductive technique. Our tool, named Tahrir, is built on top of a generic interface to SMT solvers and can be generally used for verifying SSLTL properties about infinite-state systems.

Acknowledgments

It is a pleasure to thank all those who made this thesis possible. I owe my deepest gratitude to my supervisor, Dr. Mark Aagaard, who provided me with all the technical and moral support that I needed throughout the course of my study. I am indebted to him for the countless hours he dedicated to help me develop this work. I am also thankful to him for the continuous encouragement without which I could not have completed this research.

I am also grateful to my advisory committee, Prof. Anwarul Hasan, Dr. John Thistle and Dr. Richard Trefler, and my external examiner, Dr. Clark Barrett for the invaluable comments and suggestions which help significantly improve the quality of this thesis.

I am heartily thankful to all my friends whose sincere support kept me going through difficult times. In particular, I would like to thank Khaled Hammouda, Shady Shehata, Mohammed El-Abd, Hassan Hassan, Ahmed Youssef, Ismael El-Samahy, Ayman Ismail, Nayer Wanas, Mohamed El Said, Muhammad Nummer, Wael Abdel Wahab, Mahmoud Khater, Mohamed Abu-Rahma, Mohamed Said, and Mohamed El-Dery, for being such great friends.

I would like to take this opportunity to express my love and gratitude to my family. First and foremost, I would like thank my father, Ibrahim Shehata, for always being there for me. I am also grateful to my brothers Fady, Mohamed, and Rahim for believing in me. I cannot stress how much I am thankful to this wonderful family for sticking together during times of hardship, and especially, the last few months of my mother's life.

Finally, I would like to dedicate this thesis to my late mother, Sawsan Mansour, who passed away a few months before witnessing the completion of this work. May God be merciful to her for all the sacrifices she made for our family.

To my mother

Table of Contents

List of Figures **xi**

1 Introduction **1**

1.1 Thesis Contributions 4

1.2 Thesis Outline 5

2 Inductive Verification of SSLTL **6**

2.1 Background and Related Work 7

2.1.1 State Transition Systems 7

2.1.2 Büchi Automata 9

2.1.3 Linear Temporal Logic (LTL) 12

2.1.4 LTL Verification 15

2.2 SSLTL Verification Algorithm 17

2.2.1 Translating SSLTL into Büchi Automata (*SSLTLtoBA*) 18

2.2.2 Converting Büchi Automata to Models (*BAtoModel*) 18

2.2.3 Generating Invariants from Büchi Automata (*BAtoInvar*) 19

2.2.4 Combining Models (*MergeM*) 20

2.2.5 Unfolding Models (*Unfold*) 21

2.2.6 Expanding Invariants (*Expand*) 23
2.2.7 Checking Assignments against Boolean Expressions (*Check*) 23
2.2.8 K-Step Induction over Models (*KInd*) 24
2.2.9 Verifying Models (*Verify*) 25
2.3 Soundness and Completeness of SSLTL Verification Algorithm 26
2.3.1 Proof of Theorem 2.1 31
2.4 Concluding Remarks 39
2.5 Summary 42

3 Inter-Instruction Dependency Correctness Criteria 43

3.1 Background and Related Work 44
3.1.1 Pipelining in Microprocessors 44
3.1.2 Formal Verification of Microprocessors 51
3.2 Pipeline Example: *SimPipe* 54
3.3 Parcel-Centric View of Pipelines 57
3.4 Parcel-Based Instrumentation of Pipelines 60
3.5 Parcel-Based Correctness of Pipelines 65
3.6 Specifying Inter-Parcel Correctness 69
3.7 Decomposing Inter-Parcel Correctness 75
3.7.1 Obligations 76
3.7.2 Consistency Conditions 80
3.8 Soundness of Decomposition 83
3.9 Summary 87

4 Processor Case Study 88

4.1 SSLTL Verification Tool - Tahrir 89

4.2 Processor Microarchitecture 92

4.3 Implementing the Microarchitecture 95

4.4 Verifying Inter-instruction Dependencies 98

4.4.1 Instrumentation Predicates 100

4.4.2 Lemmas and Assumptions 108

4.5 Verification Remarks 110

4.6 Summary 113

5 Conclusions 114

References 126

List of Figures

2.1 An example of a state transition system (STS) 8

2.2 An example of a Büchi automaton 11

2.3 An example of an STS-BA product 27

2.4 Proof of sketch of the corollary 29

2.5 Proof of G1 33

2.6 Proof of G1 (Continued) 34

2.7 Proof of G2 35

2.8 Proof of G2 (Continued) 36

2.9 Proof of G3 37

2.10 Proof of G3 (Continued) 38

2.11 Proof of G4 40

2.12 Proof of G4 (Continued) 41

3.1 A 5-stage pipeline 45

3.2 Sequence of instructions 47

3.3 *SimPipe* pipeline 55

3.4 Non-pipelined specification of *SimPipe* 56

3.5 Producer-consumer property 72

3.6 No-producer property 74
3.7 Decomposition tree of the producer-consumer property 76
3.8 Decomposition tree of the no-producer property 77
3.9 Sketch of decomposition proof of the producer-consumer property 85
3.10 Sketch of decomposition proof of the no-producer property 86

4.1 An out-of-order speculative microarchitecture 93
4.2 Storage elements of the processor model 96
4.3 Büchi automaton for obligation $\mathbf{Ob3a}^{\mathtt{PC}}_{\mathtt{PC}}$ 110
4.4 CPU time and memory consumption: CVC3 (Top) and UCLID (Bottom) . 112

Chapter 1

Introduction

In microprocessors, achieving an efficient utilization of the execution units is a key factor in improving performance. However, maintaining an uninterrupted flow of instructions is a challenge due to the data and control dependencies between instructions of a program. Modern microprocessors employ aggressive optimizations trying to keep their execution units busy without violating inter-instruction dependencies. Such complex optimizations may cause subtle implementation bugs that can be hard to detect using conventional simulation-based verification techniques.

It was estimated that if a bug similar to the Pentium FDIV bug* were to go undetected in the Intel® Pentium® 4 processor, it would cost Intel $12 Billion [5]. Such devastating economic effect motivates the use of formal verification approaches. Formal verification is known for its ability to discover design flaws that may not be detected using conventional verification techniques. The power of formal verification approaches lies in their exhaustive nature which enables detecting any violation of the processor specifications early in the design phase.

Formal verification is the act of using mathematical methods in proving or disproving the correctness of an *implementation* with respect to a certain *specification*. In the context

*Floating point Division (FDIV) bug, discovered in 1994, resulted in Intel's first ever chip-recall and a charge against earnings of $475 million.

of hardware systems, the term *implementation* refers to a design description at any level of the hardware abstraction hierarchy, not only the final circuit layout [21]. The term *specification* refers to the desired (correct) behavior of the design under consideration.

The most common techniques used in formal verification are:

- *Theorem Proving*: the implementation/specification relationship is treated as a theorem to be proved in the context of a proof calculus. Theorem proving tools, such as HOL [20] and ACL2 [30, 31], are used to guarantee the soundness of verification proofs. Human intervention is required to guide the verification.

- *Model Checking*: the verification is typically done by performing an exhaustive search over the implementation state-space. Model-checking tools, such as SMV [43] and FormalCheck [36], carry out such exhaustive searches automatically in order to minimize human intervention. Fully-automated model-checking techniques do not scale well with an increase in the size of the implementation and/or specification; this is known as the *state-space explosion* problem. Other model-checking techniques (*e.g.*, invariant-based) trade full-automation for scalability. For instance, with invariant-based model checking, such as in UCLID [6], the human verifier has to identify the invariants of the implementation, which is something done automatically in SMV for example.

Specifications can be formally represented either by a *high-level model* or by a set of *properties* [32]. In the first case, the verification goal is to make sure that all of the possible implementation behaviors are a subset of the specification behaviors; this is called *refinement-based* verification. In the second case, the verification goal is to make sure that all of the possible implementation behaviors satisfy the specification properties; this is called *assertion-based* verification.

With formal verification come two major challenges. First, the correctness of the implementation needs to be defined formally. Second, formal verification is often hard to apply at the scale of realistic implementations. To date, a fully-automatic verification of a realistic microprocessor is well beyond the capacity of any known formal verification tool.

This represents an open area for future improvements highly motivated by real industrial needs.

In this thesis, we present a formal verification strategy to guarantee that a microprocessor implementation preserves both data and control dependencies among instructions. Throughout our strategy, we address the two major challenges associated with formal verification: correctness and scalability.

We address the correctness challenge by specifying our correctness in the context of generic pipelines. Unlike conventional pipeline hazard rules, we make no distinction between the data and control aspects. Instead, we describe the relationship between two arbitrary instructions, first of which *produces* some data that should be *consumed* by the other, in such a way that both instructions can speculatively read their source operands, speculatively write their results, and go out of their program order during execution. In addition to supporting branch and value prediction, our correctness criteria allow the implementation to discard (squash) or replay instructions while being executed.

We address the scalability challenge in three ways: abstraction, decomposition, and induction. First, we state our inter-instruction dependency correctness criteria in terms of read and write operations without making reference to data values. Consequently, our correctness criteria can be verified for implementations with abstract datapaths, which reduces the verification complexity and enables verifying larger implementations. Second, we decompose our correctness criteria into a set of smaller obligations that are easier to verify. All these obligations can be expressed as properties within the syntactically-safe fragment of linear temporal logic (SSLTL). Third, we introduce a technique to verify SSLTL properties by induction, and prove its soundness and completeness.

To check whether an implementation satisfies an SSLTL property, we first compile the formula into a non-deterministic Büchi automaton. Then, we augment the implementation with a set of history variables representing the states of the Büchi automaton and generate an invariant representing the automaton's transition relation. Finally, we check whether the augmented implementation satisfies the invariant for both the base and inductive cases.

To demonstrate our overall strategy, we verified a term-level model of an out-of-order speculative processor. The processor model implements register renaming using a P6-style

reorder buffer and branch prediction with a hybrid (discard-replay) recovery mechanism. The verification obligations (expressed in SSLTL) are checked using a tool, named Tahrir, implementing our inductive technique. Tahrir is built on top of a generic interface to SMT solvers and can be generally used for verifying SSLTL properties about infinite-state systems.

1.1 Thesis Contributions

This thesis contains two major areas of research: verification of SSLTL properties on infinite-state systems, and the specification and verification of inter-instruction dependencies in microprocessors.

Our overall goal was to verify the correctness of microarchitectural algorithms. For this reason, we chose to use term-level models of microprocessors. Term-level models would allow us to focus on algorithms and not get lost in low-level hardware details.

The most natural approach for specifying correctness was to use LTL. In fact, all of our properties can be easily expressed in a fragment of LTL called "syntactically-safe LTL" (SSLTL), which is easier to verify compared to full LTL.

To accomplish this verification, we needed an effective approach for verifying SSLTL properties about term-level models. Verification of LTL properties generally is done by reachability analysis. Term-level models are infinite-state systems. Since reachability analysis will not terminate on infinite-state systems, our solution was to create an inductive approach that uses manually constructed invariants to restrict the state space. This approach made it possible for us to verify SSLTL properties about infinite-state systems. Though the SAL verification suite [15] has similar capabilities, the benefits of our work are a clearly documented algorithm with proof of correctness and a tool with a generic interface to SMT solver engines.

The conventional approach to the formal verification of a microprocessor is to construct a single, monolithic, correctness criterion. The verification relies on lemmas and invariants that are defined on a case-by-case basis for each pipeline. The conventional approach looks

at a state of the pipeline, which is problematic because the large number of in-flight parcels causes capacity problems in verification.

Our work provides a general definition of correctness and a general verification strategy that decomposes the top-level correctness statement into simpler obligations about data/control dependencies between parcels on individual variables. Our approach saves the effort and potential mistakes of creating custom definitions of correctness and verification strategies for each pipeline.

1.2 Thesis Outline

Chapter 2 explains our approach for the formal verification of SSLTL formulas. It describes the overall verification algorithm and shows its correctness. Chapter 3 switches the focus to the inter-parcel (instruction) correctness criteria and their decomposition. Chapter 4 sheds some light on the case study used to evaluate/illustrate the techniques presented in chapters 2 and 3. Chapter 5 summarizes the research presented in this thesis and offers some directions for future work.

Chapter 2

Inductive Verification of SSLTL

The first step in verifying a reactive system is to come up with a formal specification of the system. One of the common specification languages for reactive systems is *temporal logic*. Temporal logic comes in two varieties: *linear time* (*e.g.*, linear temporal logic (LTL) [51]) or *branching time* (*e.g.*, computational tree logic (CTL) [10]). The difference is that branching time logics can reason about multiple time lines while linear time logics are restricted to a single time line.

The ability to reason about more than one time line may suggest that branching time logics would be superior. However, other factors such as expressiveness, efficiency and intuitiveness need to be taken into consideration when choosing between the two classes of logic. For instance, neither CTL or LTL is more expressive than the other, and although CTL is more efficient (in terms of model-checking complexity), in practice, engineers found it easier to specify properties in LTL [62].

In this research, it was more intuitive for us to use LTL in specifying the inter-instruction dependency properties in chapter 3. Another factor favoring LTL was that all our properties could be expressed using a fragment of LTL called the *syntactically-safe linear temporal logic* (SSLTL), whose model-checking complexity is less than that of full LTL.

In this chapter, we focus on SSLTL. We provide the necessary background and demonstrate the related work in section 2.1. Then, in section 2.2, we introduce an algorithm that

allows us to check SSLTL properties about infinite-state systems inductively. We show the soundness and completeness of our algorithm in section 2.3. We conclude the chapter with a few remarks in section 2.4. A summary of the chapter can be found in section 2.5.

2.1 Background and Related Work

The goal of this section is to provide enough background information to help the reader understand the SSLTL verification strategy we describe in section 2.2. We start by defining two computational models: state transition systems (subsection 2.1.1) and Büchi automata (subsection 2.1.2). Then, we define the linear temporal logic and its syntactically-safe fragment SSLTL (subsection 2.1.3). Last, we conclude by demonstrating some of the work done on LTL verification (subsection 2.1.4).

2.1.1 State Transition Systems

A *state transition system* (STS) is a graph that enumerates all the states of a reactive system and describes the relationship between these states. Each state in an STS is labeled by the propositions that hold in this state. The computations of the original system are modeled as paths in the STS. We formally define an STS as follows:

Definition 2.1. (STS) A *state transition system* T is a five tuple $T = \langle AP, S, I, R, L \rangle$ where:

- AP is a set of atomic propositions.
- S is a (possibly infinite) set of states.
- $I \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is a total transition relation. R is total in the sense that for each $s \in S$, there exists $s' \in S$ such that $R(s, s')$.

- $L : S \mapsto 2^{AP}$ is a labeling function that identifies the true atomic propositions* in each state.

Example 2.1. The graph in figure 2.1 represents a state transition system $T = \langle AP, S, I, R, L\rangle$ where:

- $AP = \{a, b, c\}$
- $S = \{s_0, s_1, s_2\}$
- $I = \{s_0\}$
- $R = \{(s_0, s_0), (s_0, s_1), (s_0, s_2), (s_1, s_2), (s_2, s_0)\}$
- $L = \lambda\, s \in S.$ **if** $s = s_0$ **then** $\{a, c\}$ **elseif** $s = s_1$ **then** $\{c\}$ **else** $\{a, b\}$

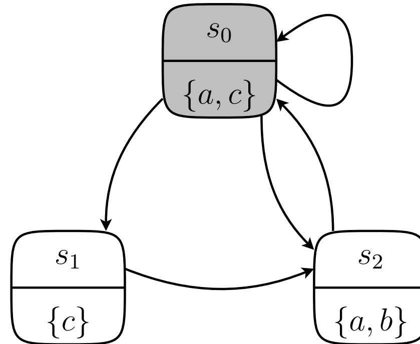

Figure 2.1: An example of a state transition system (STS)

A *path* in a state transition system $T = \langle AP, S, I, R, L\rangle$ is an infinite sequence of states $\pi = \ll\pi^0\pi^1\pi^2\ldots\gg$ where $R(\pi^i, \pi^{i+1})$ for all $i \in \mathbb{N}$. We refer to the suffix of π starting at a state π^j, for some $j \in \mathbb{N}$, as $\vec{\pi}^j$. A *run* of T is a path π_1 that starts from an initial state (*i.e.*, $\pi_1^0 \in I$). On that basis, we define the following concepts:

*A proposition is a statement that is either true or false. Atomic propositions are those propositions which cannot be represented in terms of other propositions.

Definition 2.2. (STS Concepts) Given a state transition system $T = \langle AP, S, I, R, L \rangle$:

- The set of all possible runs of T is:
  $Runs(T) = \{\pi \mid \pi^0 \in I \wedge \forall\, i \in \mathbb{N}.\ R(\pi^i, \pi^{i+1})\}$
- The set of reachable states of T is:
  $Reach(T) = \{s \mid \exists\, \pi \in Runs(T), i \in \mathbb{N}.\ s = \pi^i\}$
- The language of T is:
  $Lang(T) = \{\omega \mid \exists\, \pi \in Runs(T).\ \forall\, i \in \mathbb{N}.\ \omega^i = L(\pi^i)\}$
- The language of T restricted to a set AP' is:
  $Lang^{AP'}(T) = \{\omega \mid \exists\, \pi \in Runs(T).\ \forall\, i \in \mathbb{N}.\ \omega^i = L(\pi^i) \cap AP'\}$

Notice that, in definition 2.2, the *language* of T consists of a set of words. Each *word* is a sequence of letters. Each *letter* is a set of atomic propositions.

Next, we present the concept of *simulation* [45] as a means of comparing the behavior of state transition systems.

Definition 2.3. (Simulation) Let $T_1 = \langle AP_1, S_1, I_1, R_1, L_1 \rangle$ and $T_2 = \langle AP_2, S_2, I_2, R_2, L_2 \rangle$ be two state transition systems. We say T_2 *simulates* T_1 (denoted as $T_1 \preceq T_2$) if and only if:

$$
\begin{aligned}
& AP_2 \subseteq AP_1 \\
\wedge\ & \exists\, H \subseteq S_1 \times S_2.\ \forall\, s_1. \\
& \quad s_1 \in I_1 \Longrightarrow \exists\, s_2 \in I_2.\ H(s_1, s_2) \\
& \quad \wedge\ \forall\, s_2.\ H(s_1, s_2) \Longrightarrow L_1(s_1) \cap AP_2 = L_2(s_2) \\
& \quad \wedge\ \forall\, s_2, s_1'.\ H(s_1, s_2) \wedge R_1(s_1, s_1') \Longrightarrow \exists\, s_2'.\ H(s_1', s_2') \wedge R_2(s_2, s_2')
\end{aligned}
$$

2.1.2 Büchi Automata

A *Büchi automaton* (BA) [7] is a finite automaton that accepts infinite input sequences (*i.e.*, an ω-automaton). An input sequence is accepted if and only if the automaton visits a subset of certain states (called *accepting states*) infinitely often during its run. Büchi

automata are either *deterministic* or *non-deterministic*. Throughout this thesis, the term "Büchi automata" is used to refer to non-deterministic Büchi automata.

A simple example of a Büchi automaton is shown in figure 2.2. This automaton is non-deterministic because for instance when the automaton reaches state q_1, it is possible to accept input literals a and b, and *non-deterministically* choose to stay at q_1 or move to q_0. Conventionally, Büchi automata are defined as follows:

Definition 2.4. (BA) A *Büchi automaton* B is a five tuple $B = \langle \Sigma, Q, \dot{q}, \Delta, F \rangle$ where:

- Σ is a finite set of characters (input alphabet).
- Q is a finite set of states.
- $\dot{q} \in Q$ is the initial state.
- $\Delta \subseteq Q \times \Sigma \times Q$ is a total transition relation. Totality here means that for every $q \in Q$, there exists $q' \in Q$ and $\sigma \in \Sigma$ such that $\Delta(q, \sigma, q')$.
- $F \subseteq Q$ is the set of accepting states.

Notice that the input alphabet Σ can be defined to be the *power set* of a set of atomic propositions. In this case, every character is a subset of the atomic propositions, and should be interpreted as the *conjunction* of those atomic propositions.

Example 2.2. Suppose q_1 is the only accepting state of the Büchi automaton shown in figure 2.2. In this case, the automaton can be represented by a five tuple $\langle \Sigma, Q, \dot{q}, \Delta, F \rangle$ where:

- $\Sigma = 2^{\{a,b\}}$
- $Q = \{q_0, q_1\}$
- $\dot{q} = q_0$
- $\Delta = \{(q_0, \{a\}, q_0), (q_0, \{\}, q_1), (q_1, \{a, b\}, q_0), (q_1, \{b\}, q_1)\}$

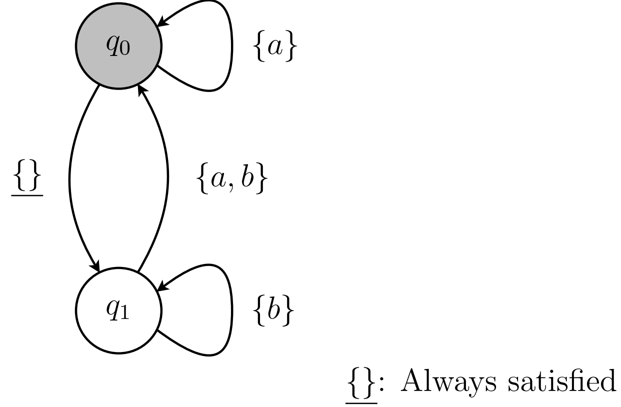

Figure 2.2: An example of a Büchi automaton

- $F = \{q_1\}$

In this thesis, we are interested in the subset of Büchi automata where every state is an accepting state[†]. This is mainly because in our SSLTL verification approach, we translate the properties into automata within that subset. Throughout the rest of the thesis, for brevity, we will drop the set of final states from the tuple representing Büchi automata in definition 2.4.

Similar to state transition systems in subsection 2.1.1, we define a *run* of a Büchi automaton $B = \langle \Sigma, Q, \dot{q}, \Delta \rangle$ as an infinite sequence of states $\pi = \ll \pi^0 \pi^1 \pi^2 \ldots \gg$ that starts from the initial state (*i.e.*, $\pi^0 = \dot{q}$) and there exists a corresponding input sequence $\sigma = \ll \sigma^0 \sigma^1 \sigma^2 \ldots \gg$ such that $\Delta(\pi^i, \sigma^i, \pi^{i+1})$ for all $i \in \mathbb{N}$. We also define the following concepts:

Definition 2.5. (BA Concepts) If $B = \langle \Sigma, Q, \dot{q}, \Delta \rangle$ is a Büchi automaton, then:

- The set of all possible runs of B is:
  $Runs(B) = \{\pi \mid \pi^0 = \dot{q} \wedge \forall\, i \in \mathbb{N}.\, \exists\, \sigma^i \in \Sigma.\, \Delta(\pi^i, \sigma^i, \pi^{i+1})\}$

[†]Of course this does not necessarily mean that the automaton accepts everything. Based on the transition relation, some inputs may not be accepted at certain states.

- The language of B is:
  $Lang(B) = \{\omega \mid \exists\, \pi \in Runs(B).\ \forall\, i \in \mathbb{N}.\ \Delta(\pi^i, \omega^i, \pi^{i+1})\}$

We also introduce the function *BAtoSTS* to syntactically transform Büchi automata to state transition systems, where every state in the automaton is represented with (possibly) multiple states in the STS, one state for each outgoing transition:

Definition 2.6. (BA to STS) Function *BAtoSTS* takes a Büchi automaton $B = \langle 2^{AP'}, Q, \dot{q}, \Delta\rangle$ and returns a state transition system $T = \langle AP, S, I, R, L\rangle$ such that:

- $AP = AP'$
- $S = \{(Z, \sigma) \mid Z \subseteq Q \wedge \sigma \subseteq AP' \wedge$
  $\exists\, q \in Z, q' \in Q, \sigma_x \subseteq \sigma.\ \Delta(q, \sigma_x, q')\}$
- $I = \{(\{\dot{q}\}, \sigma) \mid (\{\dot{q}\}, \sigma) \in S\}$
- $R = \{((Z, \sigma), (Z', \sigma')) \mid (Z, \sigma) \in S \wedge (Z', \sigma') \in S \wedge$
  $Z' = \{q' \mid \exists\, q \in Z, \sigma_x \subseteq \sigma.\ \Delta(q, \sigma_x, q')\}\}$
- $L = \lambda\,(Z, \sigma).\ \sigma$

2.1.3 Linear Temporal Logic (LTL)

Formal verification generally addresses properties with a temporal nature such as: "something *eventually* happens" or "something *never* happens". Many *temporal logics* are used for specifying such properties. We focus in this section on the Linear Temporal Logic (LTL) [51], and more specifically, a fragment of it named *syntactically-safe LTL* (SSLTL). Examples of other temporal logics are computational tree logics (CTL*, CTL and their sublogics) [10] and μ-Calculus [34].

We start by introducing the syntax of LTL. The syntax is presented in Backus Naur Form (BNF).

Definition 2.7. (LTL Syntax) The syntax of LTL formulas can be inductively described as follows:

$$\phi ::= \top \mid \bot \mid x \mid (\neg\phi) \mid (\phi \vee \phi) \mid (\phi \wedge \phi) \mid (\mathbf{X}\ \phi) \mid (\phi\ \mathbf{U}\ \phi) \mid (\phi\ \mathbf{R}\ \phi)$$

where x is any atomic proposition

In addition to regular Boolean constants and operators: True ($\top$), False ($\bot$), Negation ($\neg$), Disjunction ($\vee$) and Conjunction ($\wedge$), LTL syntax introduces three *temporal operators*: "Next" ($\mathbf{X}$), "Until" ($\mathbf{U}$), and "Release" ($\mathbf{R}$). Some other Boolean and temporal operators can be defined as syntactic sugar. Here are some examples:

- "Implies" ($\Longrightarrow$): $p_1 \Longrightarrow p_2 \equiv \neg p_1 \vee p_2$
- "Eventually" ($\mathbf{F}$): $\mathbf{F}\ p \equiv \top\ \mathbf{U}\ p$
- "Globally" ($\mathbf{G}$): $\mathbf{G}\ p \equiv \bot\ \mathbf{R}\ p$
- "Weak Until" ($\mathbf{W}$): $p_1\ \mathbf{W}\ p_2 \equiv p_2\ \mathbf{R}\ (p_1 \vee p_2)$

The semantics of LTL formulas are defined over the paths of a state transition system as follows:

Definition 2.8. (LTL Semantics) Suppose π is a path in a state transition system $T = \langle AP, S, I, R, L\rangle$. Let x be one of the atomic propositions in AP. We define the LTL *satisfaction relation* $\models$ such that:

1. $T, \pi \models \top$.
2. $\neg(T, \pi \models \bot)$.
3. $T, \pi \models x \iff x \in L(\pi^0)$.
4. $T, \pi \models \neg p_1 \iff \neg(T, \pi \models p_1)$.
5. $T, \pi \models p_1 \vee p_2 \iff T, \pi \models p_1 \vee T, \pi \models p_2$.

6. $T, \pi \models p_1 \wedge p_2 \iff T, \pi \models p_1 \wedge T, \pi \models p_2$.

7. $T, \pi \models \mathbf{X}\, p_1 \iff T, \vec{\pi}^1 \models p_1$.

8. $T, \pi \models p_1\, \mathbf{U}\, p_2 \iff \exists\, i \in \mathbb{N}.\, T, \vec{\pi}^i \models p_2 \wedge \forall\, j : 0 \leq j < i.\, T, \vec{\pi}^j \models p_1$.

9. $T, \pi \models p_1\, \mathbf{R}\, p_2 \iff \forall\, i \in \mathbb{N}.\, T, \vec{\pi}^i \models p_2 \vee \exists\, j < i.\, T, \vec{\pi}^j \models p_1$.

As a generalization of definition 2.8, we say that a system T *satisfies* a property p (written $T \models p$) if and only if p is satisfied in every *run* of T. More formally:

$$T \models p \iff \forall\, \pi \in \mathit{Runs}(T).\, T, \pi \models p$$

Two of the most important classes of properties that can be specified in LTL (or in Temporal Logics in general) are: *safety properties* and *liveness properties*. A safety property asserts that something (bad) *never happens*, while a liveness property asserts that something (good) *eventually happens*. All LTL properties in positive normal form (*i.e.*, negation is restricted to atomic propositions) constructed with the temporal operators **X** and **R** are safety properties [57, 35]. This class of LTL is referred to as the *syntactically-safe linear temporal logic*, or simply *SSLTL*.

The basic syntax of SSLTL is similar to that of definition 2.7 except that the negation is restricted to atomic propositions and the temporal operator **U** is not allowed. Only formulas in positive normal form can be constructed using the basic syntax. For better readability, we use the more flexible (yet equivalent) SSLTL syntax in definition 2.9.

Definition 2.9. (SSLTL Syntax) The SSLTL syntax is presented as follows:

$\phi ::= \top \mid \bot \mid x \mid (\neg \bar{\phi}) \mid (\phi \vee \phi) \mid (\phi \wedge \phi) \mid (\bar{\phi} \implies \phi) \mid (\mathbf{X}\ \phi) \mid (\phi\ \mathbf{R}\ \phi) \mid (\mathbf{G}\ \phi) \mid (\phi\ \mathbf{W}\ \phi)$
where:
$\bar{\phi} ::= \top \mid \bot \mid x \mid (\neg \phi) \mid (\bar{\phi} \vee \bar{\phi}) \mid (\bar{\phi} \wedge \bar{\phi}) \mid (\phi \implies \bar{\phi}) \mid (\mathbf{X}\ \bar{\phi}) \mid (\bar{\phi}\ \mathbf{U}\ \bar{\phi}) \mid (\mathbf{F}\ \bar{\phi})$
x is any atomic proposition

The syntax described in definition 2.9 does not allow an even number of negations to be applied to **U** or any other operator built on top of it (as syntactic sugar), *i.e.*, **F**. The syntax also prevents an odd number of negations to be applied to **R** or any other operator built on top of it, *i.e.*, **G** and **W**. These two restrictions ensure the formula is kept within the safe fragment of LTL.

In section 2.2, we present an algorithm for verifying SSLTL properties inductively. We use SSLTL to specify the verification obligations in chapter 3. In that chapter, we also use some Past linear temporal logic (PLTL) operators such as (*e.g.*, "Past Next" $\hat{\mathbf{X}}$, "Past Globally" $\hat{\mathbf{G}}$, and "Past Until" $\hat{\mathbf{U}}$) in specifying some intermediate proof obligations. These operators do not add expressive power to LTL [16]. However, they can help keep the properties compact and easier to read. The semantics of these PLTL operators are similar to their LTL counterparts except that they address past time as opposed to future time.

2.1.4 LTL Verification

In this section, we discuss the most common techniques for model checking linear temporal logic (LTL). The aim of these techniques is to check whether an implementation T (modeled as an STS for instance) satisfies a property p (specified in LTL), *i.e.*, to check whether $T \models p$. To be able to answer this question, p is compiled into a structure in the form of a graph (namely, a tableau or an automaton) which then can be compared against the implementation.

In the *tableau-based* approach, first the property p is used to build a tableau which is a graph (or simply an STS) that contains every path that satisfies p. Then, the tableau is composed with the implementation and the product is checked for paths that violate p. The algorithms by Lichtenstein *et al.* [38] and Clarke *et al.* [11] are two examples of the Tableau-based approach. In the first algorithm, the tableau construction is implicit while in the second the tableau is directly constructed and symbolically represented as an Ordered Binary Decision Diagram (OBDD).

The *automata-based* approach relies on the close relationship between LTL and Automata Theory which was first discussed by Wolper *et al.* [65]. Later work [63] showed

that for any given LTL property p, it is possible to construct a finite automaton B_p on infinite words that accepts exactly the set of computations that satisfy the property p. By treating the implementation system T as an automaton, the problem of checking whether $T \models p$ is transformed into an equivalent language containment problem of whether $Lang(T) \subseteq Lang B_p$.

In practice, the language containment problem is solved by checking whether $Lang(T) \cap Lang(B_{\neg p}) = \emptyset$. The first step in implementing this approach is to construct a Büchi automaton $B_{\neg p}$ for the negation of the property p. Next, a product $T \times B_{\neg p}$, whose language equals the intersection between $Lang(T)$ and $Lang(B_{\neg p})$, is computed. The last step is to check whether the language of the product $T \times B_{\neg p}$ is empty using techniques based on either performing a nested depth-first search [13, 24] or computing the maximal strongly connected components of a directed graph [58]. The property p is satisfied by the implementation T if and only if $Lang(T \times B_{\neg p})$ turns out to be empty.

The algorithm proposed by Gerth *et al.* [18] is an example of how an LTL property p is compiled into a Generalized[‡] Büchi automaton. The algorithm constructs the automaton by incrementally building a graph of nodes. Individual nodes are recursively *expanded*, *split*, or *replaced* to satisfy the subformulas of p. For representing the nodes, the algorithm uses a data structure that keeps track of which subformulas have been processed so far as well as which are left to be processed. After building the graph, the algorithm identifies the accepting states based on the nodes that are marked with subformulas of the form $\phi_1 \mathbf{U} \phi_2$. Better performing algorithms for constructing Büchi automata from LTL properties were developed by Couvreur [14], Gastin *et al.* [17], and Latvala [37].

In this thesis, we are interested in verifying properties specified using the syntactically-safe fragment of LTL (SSLTL) defined in subsection 2.1.3. Kupferman and Vardi [35] showed that for this subclass of LTL, the automaton $B_{\neg p}$ does not have to recognize all computations violating the property of p. Consequently, $B_{\neg p}$ can be computed using a more efficient approach involving the construction of an automaton on finite words. Also in the case of SSLTL, checking language emptiness for the product $T \times B_{\neg p}$ is reduced to

[‡]A Generalized Büchi automaton (GBA) is defined the same as a regular Büchi automaton except that a GBA can have multiple sets of accepting states. A word is recognized by a GBA if and only if at least one state from every accepting set is visited infinitely often.

invariance checking as opposed to checking cycles in the case of LTL.

In our strategy, we first construct a Büchi automaton B_p for the SSLTL property p itself as opposed to its negation. Then through induction, we verify that the product system satisfies an invariant about the states of B_p. For constructing B_p, we follow a straightforward approach based on that of Gerth *et al.* [18]. The difference is that in our case all the states of B_p are considered accepting states since p is syntactically-safe. The details of our strategy can be found in section 2.2.

2.2 SSLTL Verification Algorithm

In this section, we introduce an algorithm for verifying SSLTL properties inductively. The algorithm takes (among other inputs) a model and an SSLTL property, and returns true if and only if the model satisfies the property. The algorithm also takes a number representing the depth of the induction and a Boolean expression to use in strengthening the inductive invariant.

Before describing our algorithm, we first define what we mean by a *model*. The word "model" refers to the source code of the system. Although models are finite in size, they may describe infinite-state and/or non-deterministic systems. The purpose of a model is to represent the behavior of a system using a set of variables and expressions over those variables. The variables capture the state of the system and the expressions describe how the state evolves over time. A model is formally defined as follows:

Definition 2.10. (Model) Let E be a set of expressions defined by a given grammar. A *Model* M over E is a quadruple $M = \langle V, \dot{A}, \ddot{A}, \bar{A} \rangle$ where:

- V is a finite set of variables over possibly infinite domains.
- $\dot{A} \subseteq V \times E$ is the set of initial-state assignments.
- $\ddot{A} \subseteq V \times E$ is the set of next-state assignments.
- $\bar{A} \subseteq V \times E$ is a set of combinational assignments.

Example 2.3. The state transition system T in example 2.1 can be encoded by a model $M = \langle V, \dot{A}, \ddot{A}, \bar{A} \rangle$ where:

- $V = \{s, a, b, c\}$
- $\dot{A} = \{(s, S_0)\}$
- $\ddot{A} = \{(s, \textbf{if } s = S_0 \textbf{ then } S_0|S_1|S_2 \textbf{ elseif } s = S_1 \textbf{ then } S_2 \textbf{ else } S_0)\}$
- $\bar{A} = \{(a, s = S_0 \vee s = S_2), (b, s = S_2), (c, s = S_0 \vee s = S_1)\}$

For the remainder of this section (subsections 2.2.1-2.2.9), we present the set of functions used in our algorithm where the main function, called *Verify*, is presented last.

2.2.1 Translating SSLTL into Büchi Automata (*SSLTLtoBA*)

Function *SSLTLtoBA* compiles an SSLTL property into a Büchi automaton. The function implements the basic Büchi automata construction algorithm by Gerth *et al.* [18] (described in subsection 2.1.4), with the exception that all the states of the constructed automaton are considered accepting states.

Example 2.4. For an SSLTL property $p = \mathbf{G}\ (a\ \mathbf{W}\ (\mathbf{X}\ b))$, the corresponding Büchi automaton $B = \mathit{SSLTLtoBA}(p)$ is shown in figure 2.2.

2.2.2 Converting Büchi Automata to Models (*BAtoModel*)

The purpose of function *BAtoModel* is to generate a model from a Büchi automaton B. Every state in the automaton is represented by a Boolean (state) variable in the generated model. The variable associated with the initial state of the automaton is initialized to 1 while all the other variables are initialized to 0 (line 7). The next values of the variables are defined as Boolean expressions encoding the transition relation of the automaton (line 9). The automaton is completely represented by the state variables and hence no combinational variables are added to the generated model.

Function `BAtoM`(B : BA)

Define: $\langle \Sigma, Q, \dot{q}, \Delta \rangle \equiv B$

1 $V := Q$;
2 $\dot{A} := \{\}$;
3 $\ddot{A} := \{\}$;
4 $\bar{A} := \{\}$;
5 **foreach** $q \in Q$ **do**
6 $\quad e_{in} := 0$;
7 $\quad$ **if** $q = \dot{q}$ **then** $\dot{A} := \dot{A} \cup \{(q, 1)\}$ **else** $\dot{A} := \dot{A} \cup \{(q, 0)\}$;
8 $\quad$ **foreach** $(q_1, \sigma_1, q_2) \in \Delta$ **do**
9 $\quad\quad$ **if** $q = q_2$ **then** $e_{in} := e_{in} \vee q_1 \wedge \sigma_1$;
10 $\quad$ **end**
11 $\quad \ddot{A} := \ddot{A} \cup \{(q, e_{in})\}$;
12 **end**
13 **return** $\langle V, \dot{A}, \ddot{A}, \bar{A} \rangle$;

Example 2.5. If the automaton generated in example 2.2 is to be passed to function *BAtoModel*, the function would return a model $M = \langle V, \dot{A}, \ddot{A}, \bar{A} \rangle$ where:

- $V = \{q_0, q_1\}$
- $\dot{A} = \{(q_0, 1), (q_1, 0)\}$
- $\ddot{A} = \{(q_0, (q_0 \wedge a) \vee (q_1 \wedge a \wedge b)), (q_1, q_0 \vee (q_1 \wedge b))\}$[§]
- $\bar{A} = \{\}$

2.2.3 Generating Invariants from Büchi Automata (*BAtoInvar*)

Function *BAtoInvar* produces a Boolean expression describing the states and transitions of a Büchi automaton B. The output Boolean expression is a conjunction of a set of clauses.

[§]Symbols a and b are free variables.

Each clause encodes a single state (as a Boolean variable) and all its outgoing transitions (as a Boolean expression). The generated Boolean expression can be true if and only if at least one state and one transition going out of it are satisfied.

Function `BAtoI`(B : BA)

Define: $\langle \Sigma, Q, \dot{q}, \Delta \rangle \equiv B$

1 $e := 0$;
2 **foreach** $q \in Q$ **do**
3     $e_{out} := 0$;
4   **foreach** $(q_1, \sigma_1, q_2) \in \Delta$ **do**
5   **if** $q = q_1$ **then** $e_{out} := e_{out} \vee \sigma_1$;
6   **end**
7     $e := e \vee q \wedge e_{out}$;
8 **end**
9 **return** e;

Example 2.6. The invariant generated by function *BAtoInvar* for the automaton from example 2.2 is:
$e = (q_0 \wedge (a \vee \texttt{True})) \vee (q_1 \wedge (b \vee (a \wedge b))) = q_0 \vee (q_1 \wedge b)$

2.2.4 Combining Models (*MergeM*)

As its name suggests, function *MergeM* combines two models M_1 and M_2 into one. This is done simply by constructing the union between each component from model M_1 with the corresponding component from model M_2.

Function `Merge`(M_1 : *Model*, M_2 : *Model*)

Define: $\langle V_1, \dot{A}_1, \ddot{A}_1, \bar{A}_1 \rangle \equiv M_1$
Define: $\langle V_2, \dot{A}_2, \ddot{A}_2, \bar{A}_2 \rangle \equiv M_2$

1 **return** $\langle V_1 \cup V_2, \dot{A}_1 \cup \dot{A}_2, \ddot{A}_1 \cup \ddot{A}_2, \bar{A}_1 \cup \bar{A}_2 \rangle$;

Example 2.7. Function *MergeM* combines the two models from example 2.3 and example 2.5 into a model $M = \langle V, \dot{A}, \ddot{A}, \bar{A} \rangle$ where:

- $V = \{s, a, b, c, q_0, q_1\}$
- $\dot{A} = \{(s, S_0), (q_0, 1), (q_1, 0)\}$
- $\ddot{A} = \{(s, \textbf{if } s = S_0 \textbf{ then } S_0|S_1|S_2 \textbf{ elseif } s = S_1 \textbf{ then } S_2 \textbf{ else } S_0),$
  $(q_0, q_0 \wedge a \vee q_1 \wedge a \wedge b),$
  $(q_1, q_0 \vee q_1 \wedge b)\}$
- $\bar{A} = \{(a, s = S_0 \vee s = S_2), (b, s = S_2), (c, s = S_0 \vee s = S_1)\}$

2.2.5 Unfolding Models (*Unfold*)

The goal of function *Unfold* is to represent the values of the variables of a model M when it runs for a given number of steps k. The values of the model variables at any given simulation step i are represented by a fresh set of variables V^i. For each step, the assignments associated with the model variables are replicated and rewritten using the fresh variables.

Example 2.8. Unfolding the combined model from example 2.7 for one step ($k = 1$) produces an output (V, A) where:

- $V = \{s^0, q_0^0, q_1^0, a^0, b^0, c^0,$
  $s^1, q_0^1, q_1^1, a^1, b^1, c^1\}$
- $A = \{(s^0, S_0),$
  $(q_0^0, 1), (q_1^0, 0),$
  $(a^0, s^0 = S_0 \vee s^0 = S_2), (b^0, s^0 = S_2), (c^0, s^0 = S_0 \vee s^0 = S_1),$
  $(s^1, \textbf{if } s^0 = S_0 \textbf{ then } S_0|S_1|S_2 \textbf{ elseif } s^0 = S_1 \textbf{ then } S_2 \textbf{ else } S_0),$
  $(q_0^1, q_0^0 \wedge a^0 \vee q_1^0 \wedge a^0 \wedge b^0), (q_1^1, q_0^0 \vee q_1^0 \wedge b^0),$
  $(a^1, s^1 = S_0 \vee s^1 = S_2), (b^1, s^1 = S_2), (c^1, s^1 = S_0 \vee s^1 = S_1)\}$

Function `Unfold`(M : *Model*, k : $\mathbb{N}$)

Define: $\langle V, \dot{A}, \ddot{A}, \bar{A}\rangle \equiv M$

Define: $V^i \equiv \{v^i \mid v \in V\}$

```
1  V_r := V^0;
2  A_r := {};
3  foreach (v, e) ∈ Ȧ ∪ Ā do
4      A_r := A_r ∪ {(v^0, e[V\V^0])};
5  end
6  for j := 1 to k do
7      V_r := V_r ∪ V^j;
8      foreach (v, e) ∈ Ä do
9          A_r := A_r ∪ {(v^j, e[V\V^{j-1}])};
10     end
11     foreach (v, e) ∈ Ā do
12         A_r := A_r ∪ {(v^j, e[V\V^j])};
13     end
14 end
15 return (V_r, A_r);
```

2.2.6 Expanding Invariants (***Expand***)

Function *Expand* rewrites an invariant e for the purpose of induction over a given number of steps k. As in function *Unfold*, a fresh set of variables is used to represent the values of the model variables at each step. Using these fresh variables, the function creates an instance of the invariant for each step and returns the conjunction of all these instances.

Function `Expand`(V : *VSet*, e : *BExpr*, k : $\mathbb{N}$)

Define: $V^i \equiv \{v^i \mid v \in V\}$

1 $e_r := 1$;
2 **for** $j := 0$ **to** k **do**
3 $\quad e_r := e_r \wedge e[V \backslash V^j]$;
4 **end**
5 **return** e_r;

Example 2.9. Expanding the invariant from example 2.6 for one step ($k = 1$) produces a Boolean expression e where:
$e = q_0^0 \vee (q_1^0 \wedge b^0) \quad \wedge \quad q_0^1 \vee (q_1^1 \wedge b^1)$

2.2.7 Checking Assignments against Boolean Expressions (***Check***)

The purpose of function *Check* is to determine whether a set of assignments A satisfy a Boolean expression e. The function generates a formula e_r with an implication where the antecedent is the conjunction of the assignments in A and the consequent is the Boolean expression e. Then, the function returns true if and only if the generated formula e_r is valid. Notice that function *Check* can be viewed as a complete *decision procedure* since the function terminates (*i.e.*, returns a value of true or false) for all expressions in the supported grammar.

Function `Check`(V : *VSet*, A : *ASet*, e : *BExpr*)

1 $e_r := 1$;
2 **foreach** $(v_1, e_1) \in A$ **do**
3     $e_r := e_r \wedge (v_1 = e_1)$;
4 **end**
5 $e_r := e_r \implies e$;
6 **if** $e_r = 0$ **then return false**;
7 **return true**;

2.2.8 K-Step Induction over Models (*KInd*)

The goal of function *KInd* is to check whether a model M satisfies an invariant e by induction over k steps. In the base case, the model M is unfolded (from its initial state) for $k-1$ steps and the invariant is expanded for the same number of steps. While in the inductive case, the model M is unfolded for k steps starting from an arbitrary state (obtained by ignoring the initial-state assignments) while the induction hypothesis is formed by expanding the invariant for $k-1$ steps. The function returns true if and only if the invariant is satisfied in both the base and inductive cases.

Function `KInd`(M : *Model*, e : *BExpr*, k : $\mathbb{N}^+$)

Define: $\langle V, \dot{A}, \ddot{A}, \bar{A} \rangle \equiv M$

`/* Base Case */`
1 $(V_b, A_b) := \mathit{Unfold}(M, k-1)$;
2 $e_b := \mathit{Expand}(V, e, k-1)$;
3 **if** $\mathit{Check}(V_b, A_b, e_b) =$ **false then return false**;
`/* Inductive Case */`
4 $(V_i, A_i) := \mathit{Unfold}(\langle V, \{\}, \ddot{A}, \bar{A} \rangle, k)$;
5 $e_i := \mathit{Expand}(V, e, k-1)$;
6 **if** $\mathit{Check}(V_i, A_i, e_i \implies e[V \backslash V^k]) =$ **false then return false**;
7 **return true**;

2.2.9 Verifying Models (*Verify*)

Function *Verify* is the core of our SSLTL verification algorithm. The function checks whether a model M satisfies an SSLTL property p by induction over a given number of steps k. In addition to M, p, and k, the function takes an input Boolean expression e for the purpose of strengthening the invariant during induction.

Function *Verify* starts by translating the property p into an automaton B_p. The automaton B_p is then compiled into a model M_p and an invariant e_p. The generated model M_p is combined together with the input model M into a new model M_a, which we refer to as the *augmented model*. Then, function *Verify* checks whether the augmented model M_a satisfies the invariant e_p using k-step induction. To keep induction within the reachable state space of the augmented model, the invariant e_p is strengthened using a Boolean expression e.

Function *Verify* returns true if and only if the induction shows that the augmented model M_p satisfies the strengthened invariant $e_p \wedge e$ for the given values of e and k. In section 2.3, we show the soundness and completeness of this strategy. For the soundness, we prove that the input model M satisfies the SSLTL property p if *Verify* returns true for certain values of e and k. For the completeness, we show that if M satisfies p, there exist values for e and k that would cause *Verify* to return true.

Function `Verify`(M : *Model*, p : *SSLTL*, e : *BExpr*, k : $\mathbb{N}^+$)

Define: $\langle V, \dot{A}, \ddot{A}, \bar{A} \rangle \equiv M$
Define: $\langle V_a, \dot{A}_a, \ddot{A}_a, \bar{A}_a \rangle \equiv M_a$

1. $B_p := SSLTLtoBA(p)$;
2. $M_p := BAtoModel(B_p)$;
3. $e_p := BAtoInvar(B_p)$;
4. $M_a := MergeM(M, M_p)$;
5. $r := KInd(M_a, e_p \wedge e, k)$;
6. **return** r;

2.3 Soundness and Completeness of SSLTL Verification Algorithm

In this section, we prove that the SSLTL verification algorithm in section 2.2 is both *sound* and *complete*. Before presenting the proof, we first define a function, named *MergeS*, that combines a state transition system with a Büchi automaton.

Definition 2.11. (STS-BA Product) Function *MergeS* takes a state transition system $T = \langle AP, S, I, R, L\rangle$ and a Büchi automaton $B = \langle 2^{AP'}, Q, \dot{q}, \Delta\rangle$ and returns a state transition system $T_1 = \langle AP_1, S_1, I_1, R_1, L_1\rangle$ such that:

- $AP_1 = AP$
- $S_1 = S \times 2^Q$
- $I_1 = \{(s, \{\dot{q}\}) \mid s \in I\}$
- $R_1 = \{((s, Z), (s', Z')) \mid R(s, s') \wedge$
  $Z' = \{q' \mid \exists\, q \in Z, \sigma \subseteq L(s).\ \Delta(q, \sigma, q')\}\}$
- $L_1 = \lambda\,(s, Z).\ L(s)$

Here is an example to illustrate the purpose of function *MergeS*:

Example 2.10. Consider the state transition system T from example 2.1 and the Büchi automaton B from example 2.2 (assuming both q_0 and q_1 are final states). A state transition system T_1 constructed by combining T and B such that $T_1 = \mathit{MergeS}(T, B)$, is shown in figure 2.3. T_1 represents T and B run in parallel and synchronized based on common labels.

The core of our proof is to show that: the problem of *proving language containment* between a state transition system T and a Büchi automaton can be transformed into a problem of *verifying an invariant* about the states of the product state transition system $T_1 = \mathit{MergeS}(T, B)$. We state this result in theorem 2.1. The theorem also includes the equivalent problem in the *simulation* domain for sake of completeness. A detailed proof of theorem 2.1 can be found in subsection 2.3.1.

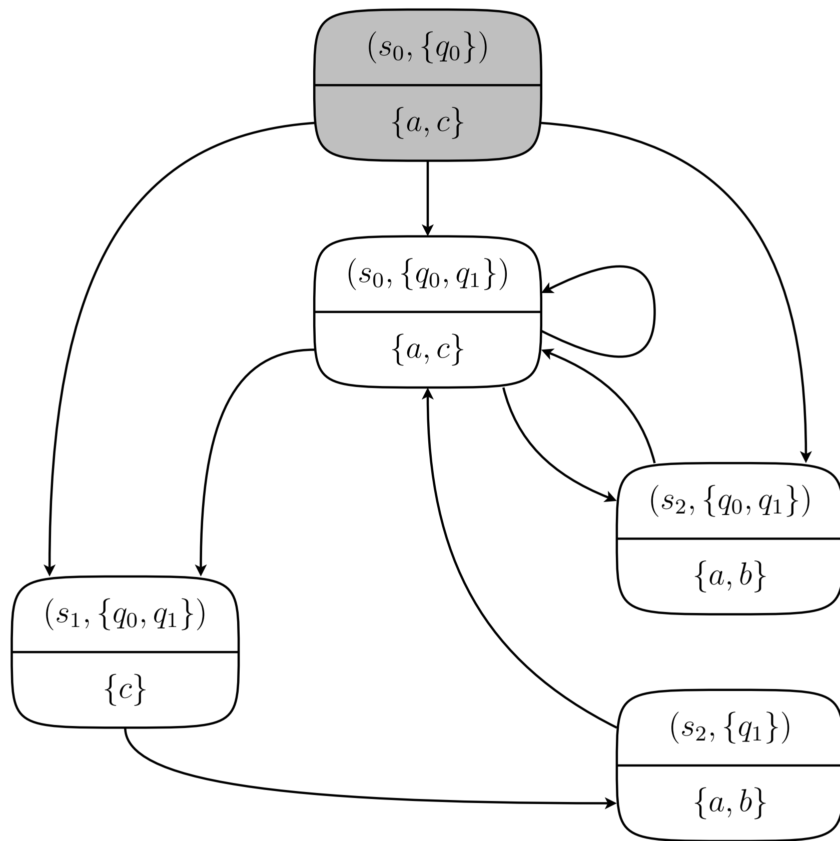

Figure 2.3: An example of an STS-BA product

Theorem 2.1. *Given a state transition system* $T = \langle AP, S, I, R, L\rangle$ *and a finite automaton* $B = \langle \Sigma, Q, \dot{q}, \Delta\rangle$ *where* $\Sigma = 2^{AP'}$ *for a set of atomic propositions* AP' *where* $AP' \subseteq AP$, *if* T_1 *and* T_2 *are defined such that* $T_1 = \mathit{MergeS}(T, B)$ *and* $T_2 = \mathit{BAtoSTS}(B)$, *then the following holds:*

$$\begin{aligned} &\forall\, s.\ (s, \{\}) \notin \mathit{Reach}(T_1) \\ &\iff \mathit{Lang}^{AP'}(T) \subseteq \mathit{Lang}(B) \\ &\iff T_1 \preceq T_2 \end{aligned}$$

Theorem 2.1 is used to show that applying our SSLTL verification algorithm (represented by function *Verify*) on a model M and an SSLTL property is equivalent to verifying that M (after being compiled to an STS using function *MtoSTS*) satisfies p. We formally state this result in the following corollary:

Corollary. *For any given model* $M = \langle V, \dot{A}, \ddot{A}, \bar{A}\rangle$ *and SSLTL property* p, *the following holds:*

$$(\exists\, e, k.\ \mathit{Verify}(M, p, e, k)) \quad \iff \quad \mathit{MtoSTS}(M) \models p$$

The soundness and completeness of our SSLTL verification strategy (function *Verify*) are captured in this corollary by the right implication ($\Longrightarrow$) and the left implication ($\Longleftarrow$) respectively. For the completeness result, we assume that the reachable states of the augmented model can be described by an expression in the grammar supported by the (complete) decision procedure represented by function *Check*.

In reality, even if the reachable states cannot be described by an expression within the grammar, an over approximation (in the form of an invariant) might be sufficient for the decision procedure to terminate. In our experience, the fact that reachable states may not be expressible in the grammar of the decision procedure has no practical impact. The grammars supported by modern decision procedures (*e.g.*, SMT solvers) are sufficiently general that we are able to write invariants strong enough to carry out the verification.

To prove the corollary, we start from the left hand side and apply a set of transformations until reaching the right hand side. The proof, sketched in figure 2.4, boils down to five main steps:

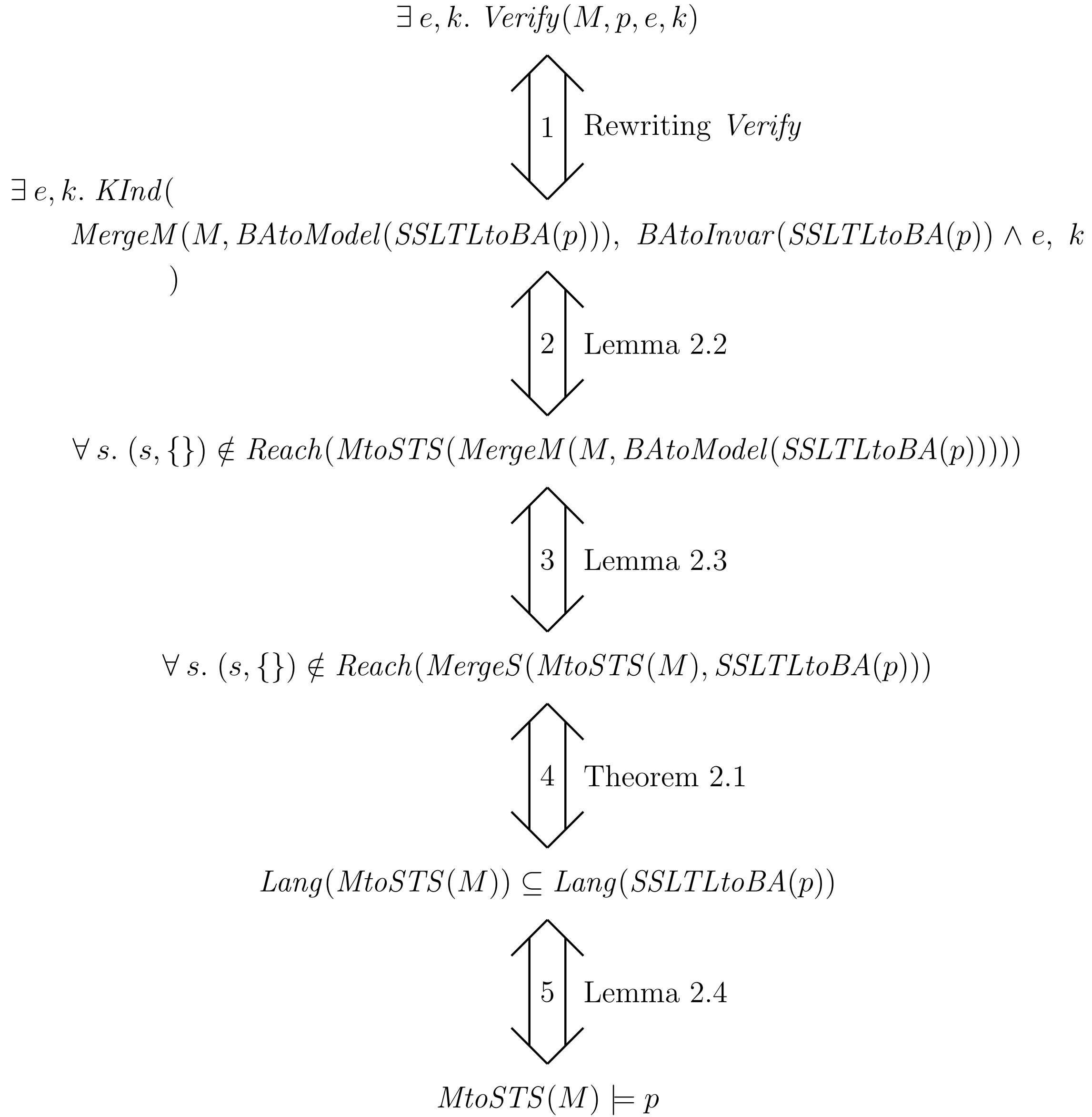

Figure 2.4: Proof of sketch of the corollary

1. Function *Verify* is rewritten using its definition from subsection 2.2.9.

2. K-Induction on models is expressed as reachability on state transition systems by substituting $\lambda\,(s,Q).\ Q \neq \{\}$ for predicate[¶] x in the following lemma:

   Lemma 2.2. *Given a model M and a predicate x, the following equivalence holds:*

   $$(\exists\, e,k.\ \mathit{KInd}(M, x \wedge e, k)) \iff \forall\, s_1 \in \mathit{Reach}(\mathit{MtoSTS}(M)).\ x(s_1)$$

   The proof of lemma 2.2 is done by setting k to one (*i.e.*, 1-step induction) and choosing e to be the Boolean expression that represents all reachable states of the STS that corresponds to M (*i.e.*, $\mathit{MtoSTS}(M)$). As mentioned earlier, we assume that reachable states can be expressed by an expression in the grammar.

3. The act of merging two models is transformed to an equivalent merge between an STS and a BA using the following lemma:

   Lemma 2.3. *The following equivalence holds true for any model M and Büchi automaton B:*

   $$\mathit{MtoSTS}(\mathit{MergeM}(M, \mathit{BAtoModel}(B))) \iff \mathit{MergeS}(\mathit{MtoSTS}(M), B)$$

   The proof of lemma 2.3 takes the form of a commuting diagram.

4. Theorem 2.1 is used to link reachability to language containment.

5. The relationship between language containment and SSLTL satisfaction is established through lemma 2.4 which capture the correctness of the Büchi automata construction algorithm.

   Lemma 2.4. *Given a state transition system T and an SSLTL property p, $T \models p$ if and only if $\mathit{Lang}(T) \subseteq \mathit{Lang}(\mathit{SSLTLtoBA}(p))$.*

[¶] A predicate is a function which evaluates to either true or false. Hence predicates can be treated as Boolean expressions. For this reason, we are able to apply the Boolean operator $\wedge$ to x in lemma 2.2

2.3.1 Proof of Theorem 2.1

The proof of theorem 2.1 is broken down into four goals, one per each implication. Before stating the goals, we first mention the premises of the proof:

P1. $T = \langle AP, S, I, R, L \rangle$ is a state transition system

P2. $B = \langle \Sigma, Q, \dot{q}, \Delta \rangle$ is an automaton

P3. $AP' \subseteq AP$

P4. $\Sigma = 2^{AP'}$

P5. $T_1 = \mathit{MergeS}(T, B)$

P6. $T_2 = \mathit{BAtoSTS}(B)$

Given the six premises, the goals can be stated as follows:

G1. $\forall\, s.\ (s, \{\}) \notin \mathit{Reach}(T_1) \Longrightarrow \mathit{Lang}^{AP'}(T) \subseteq \mathit{Lang}(B)$

G2. $\forall\, s.\ (s, \{\}) \notin \mathit{Reach}(T_1) \Longleftarrow \mathit{Lang}^{AP'}(T) \subseteq \mathit{Lang}(B)$

G3. $\forall\, s.\ (s, \{\}) \notin \mathit{Reach}(T_1) \Longrightarrow T_1 \preceq T_2$

G4. $\forall\, s.\ (s, \{\}) \notin \mathit{Reach}(T_1) \Longleftarrow T_1 \preceq T_2$

Our general strategy for proving each of these goals is to assume the precedent of the implication holds, and show that the consequent has to hold as a consequence.

- Proof of goal G1:

  Figures 2.5 and 2.6 illustrate the proof of goal G1. We assume that none of the states $(s, \{\})$, for all values of s, can be reached in T_1 (step 1), and show that the language of T has to be a subset of the language of B. To prove such language containment, we show that for any given word w_T in the language of T (step 2), w_T has to be in the language of B as well (step 29).

Suppose s_T is one of the runs of T that generate w_T (step 5). A sequence Z_T is defined to keep track of all B-states which are visited if B is to generate w_T (step 7). s_T and Z_T are combined to construct a sequence s_{T_1} (step 8). Then s_{T_1} is shown to be a run of T_1 (step 18). Since states $(s, \{\})$ are not reachable in T_1, then Z_T^i is non-empty for each value of i (step 20). That guarantees the existence of a run of B that generates the word w_T (step 28). Hence, w_T is in the language of B (step 29).

- Proof of G2:

  The proof of goal G2 is presented in figures 2.7 and 2.8. We assume that every word in the language of T has to be in language of B (step 1) and prove that for any value of s, none of the states $(s, \{\})$ is reachable in T_1 (step 27). That is realized by showing that any state s_x that is reachable in T_1 (step 2) cannot have the empty set as its second component (step 26).

  Suppose that s_x is located on a run of T_1 named s_{T_1} (step 5) and a sequence Z_T is defined such that for each i, Z_T^i equals the set of B-states associated with $s_{T_1}^i$ (step 7). By definition of T, the word produced by s_{T_1} (named w_k) has to be in the language of T (step 11) and hence in the language of B as well (step 12).

  Let q_B be one of the runs of B that generates w_T (step 14). By induction, we show that for any integer i, Z_T^i has to contain at least the state q_B^i (step 22). That guarantees that s_x cannot have the empty set as its second component (step 26).

- Proof of G3:

  Figures 2.9 and 2.10 outline the proof of goal G3. We assume that for all values of s, none of the states $(s, \{\})$ can be reached in T_1 (step 1), and show that T_2 simulates T_1 (step 37). To prove simulation, we define a binary relation $\hat{H}$ such that it contains every pair $((s, Z), (Z, L(s) \cap AP'))$ if and only if (s, Z) is among the reachable states of T_1 (step 2) and show that $\hat{H}$ is a simulation relation between T_1 and T_2. In other words, we show that $\hat{H}$ is a binary relation between S_1 and S_2, and by definition satisfies the three simulation conditions: initial, invariant, and inductive (step 36).

  To show that $\hat{H}$ is a binary relation between T_1 and T_2 states, let $((s_T, Z_T), (Z_B, \sigma_B))$ be a pair of arbitrary states that belong to $\hat{H}$ (step 3). By definition of $\hat{H}$, (s_T, Z_T)

1	$\forall\, s.\ (s, \{\}) \notin \mathit{Reach}(T_1)$	
2	$w_T \in \mathit{Lang}^{AP'}(T)$	
3	$\forall\, \pi \in \mathit{Runs}(T_1), i \in \mathbb{N}, s.\ (s, \{\}) \neq \pi^i$	1
4	$\exists\, \pi \in \mathit{Runs}(T).\ \forall\, i \in \mathbb{N}.\ w_T^i = L(\pi^i) \cap AP'$	2
5	s_T $\quad s_T \in \mathit{Runs}(T) \wedge \forall\, i \in \mathbb{N}.\ w_T^i = L(s_T^i) \cap AP'$	4
6	$s_T^0 \in I \wedge \forall\, i \in \mathbb{N}.\ R(s_T^i, s_T^{i+1})$	5
7	$Z_T \equiv \lambda\, i \in \mathbb{N}.\ \{q' \mid i = 0 \wedge q' = \dot{q} \vee i \neq 0 \wedge \exists\, q \in Z_T^{i-1}.\ \Delta(q, w_T^{i-1}, q')\}$	
8	$s_{T_1} \equiv \lambda\, i \in \mathbb{N}.\ (s_T^i, Z_T^i)$	
9	$s_{T_1}^0 = (s_T^0, \{\})$	7, 8
10	$s_{T_1}^0 \in I_1$	6, 9
11	$j \in \mathbb{N}$	
12	$s_{T_1}^j = (s_T^j, Z_T^j) \wedge s_{T_1}^{j+1} = (s_T^{j+1}, Z_T^{j+1})$	8, 11
13	$R(s_T^j, s_T^{j+1})$	4, 9
14	$Z_T^{j+1} = \{q' \mid \exists\, q \in Z_T^j.\ \Delta(q, w_T^j, q')\}$	7, 11
15	$w^j \subseteq L(s_T^j)$	5
16	$R_1(s_{T_1}^j, s_{T_1}^{j+1})$	12-15
17	$\forall\, i \in \mathbb{N}.\ R_1(s_{T_1}^i, s_{T_1}^{i+1})$	11, 16
18	$s_{T_1} \in \mathit{Runs}(T_1)$	10, 17
19	$\forall\, i \in \mathbb{N}, s.\ (s, \{\}) \neq s_{T_1}^i$	3, 18
20	$\forall\, i \in \mathbb{N}.\ Z_T^i \neq \{\}$	8, 19

Figure 2.5: Proof of G1

Line	Formula	Justification
21	$j \in \mathbb{N}$	
22	$Z_T^j \neq \{\}$	20, 21
23	$\exists\, q' \in Z_T^j.\ j = 0 \wedge q' = \dot{q}$ $\vee\ j \neq 0 \wedge \exists\, q \in Z_T^{j-1}.\ \Delta(q, w_T^{j-1}, q')$	7, 22
24	q_B^j | $q_B^j \in Z_T^j \wedge j = 0 \wedge q_B^j = \dot{q}$ $\vee\ j \neq 0 \wedge q_B^{j-1} \in Z_T^{j-1} \wedge \Delta(q_B^{j-1}, w_T^{j-1}, q_B^j)$	23
25	$q_B^0 = \dot{q} \wedge j \neq 0 \Longrightarrow \Delta(q_B^{j-1}, w_T^{j-1}, q_B^j)$	24
26	$\exists\, \pi.\ \pi^0 = \dot{q} \wedge j \neq 0 \Longrightarrow \Delta(\pi^{j-1}, w_T^{j-1}, \pi^j)$	23-25
27	$\exists\, \pi.\ \pi^0 = \dot{q} \wedge \forall\, i \in \mathbb{N}.\ \Delta(\pi^i, w_T^i, \pi^{i+1})$	21, 26
28	$\exists\, \pi \in \mathit{Runs}(B).\ \forall\, i \in \mathbb{N}.\ \Delta(\pi^i, w_T^i, \pi^{i+1})$	27
29	$w_T \in \mathit{Lang}(B)$	28

Figure 2.6: Proof of G1 (Continued)

Line	Formula	Justification
1	$\forall\, w.\ w \in \mathit{Lang}^{AP'}(T) \Longrightarrow w \in \mathit{Lang}(B)$	
2	$s_x \in \mathit{Reach}(T_1)$	
3	$\exists\, \pi \in \mathit{Runs}(T_1), i \in \mathbb{N}.\ s_x = \pi^i$	2
4	$\exists\, \pi.\ \pi^0 \in I_1 \wedge \forall\, l \in \mathbb{N}.\ R_1(\pi^l, \pi^{l+1})$ $\wedge \exists\, i \in \mathbb{N}.\ s_x = \pi^i$	3
5	s_{T_1} $s^0_{T_1} \in I_1 \wedge \forall\, l \in \mathbb{N}.\ R_1(s^l_{T_1}, s^{l+1}_{T_1})$ $\wedge \exists\, i \in \mathbb{N}.\ s_x = s^i_{T_1}$	4
6	$s_T \equiv \lambda\, i \in \mathbb{N}.\ (\lambda\, (s, Z).\ s)\ s^i_{T_1}$	
7	$Z_T \equiv \lambda\, i \in \mathbb{N}.\ (\lambda\, (s, Z).\ Z)\ s^i_{T_1}$	
8	$s^0_T \in I \wedge \forall\, l \in \mathbb{N}.\ R(s^l_T, s^{l+1}_T)$	5, 6
9	$s_T \in \mathit{Runs}(T)$	8
10	$w_T \equiv \lambda\, i \in \mathbb{N}.\ L_1(s^i_{T_1}) \cap AP'$	
11	$w_T \in \mathit{Lang}^{AP'}(T)$	9, 10
12	$w_T \in \mathit{Lang}(B)$	1, 11
13	$\exists\, \pi \in \mathit{Runs}(B).\ \forall\, i \in \mathbb{N}.\ \Delta(\pi^i, w^i_T, \pi^{i+1})$	12
14	q_B $q_B \in \mathit{Runs}(B) \wedge \forall\, i \in \mathbb{N}.\ \Delta(q^i_B, w^i_T, q^{i+1}_B)$	13
15	$q^0_B = \dot{q} \wedge \forall\, i \in \mathbb{N}.\ \Delta(q^i_B, w^i_T, q^{i+1}_B)$	14
16	$q^0_B \in Z^0_T$	5, 7, 15
17	$j \in \mathbb{N} \wedge q^j_B \in Z^j_T$	
18	$R_1((s^j_T, Z^j_T), (s^{j+1}_T, s^{j+1}_T))$	5-7,17
19	$w^j_T \subseteq L(s^j_T)$	10, 17
20	$\Delta(q^j_B, w^j_T, q^{j+1}_B)$	15, 17

Figure 2.7: Proof of G2

Step	Statement	Justification
21	$q_B^{j+1} \in Z_T^{j+1}$	17-20
22	$\forall\, i \in \mathbb{N}.\ q_B^i \in Z_T^i$	16, 17, 21
23	$\forall\, i \in \mathbb{N}.\ \exists\, q.\ q \in Z_T^i$	13, 14, 22
24	$\forall\, i \in \mathbb{N}.\ Z_T^i \neq \{\}$	23
25	$\forall\, s.\ s_x \neq (s, \{\})$	5, 7, 24
26	$\forall\, s.\ s_x \neq (s, \{\})$	4, 5, 25
27	$\forall\, s.\ (s, \{\}) \notin \mathit{Reach}(T_1)$	2, 26

Figure 2.8: Proof of G2 (Continued)

is a state of T_1 (step 4). Also, (Z_B, σ_B) has to be a state of T_2 (step 12) because Z_B is a subset of Q, σ_B is a subset of Σ (step 11), and there exists at least one B-state in Z_B that has an outgoing transition whose label is a subset of σ_B (step 10).

Next, we show that the simulation conditions hold. Suppose that (s_T, Z_T) is an arbitrary state of T_1. To prove the initial condition of simulation, we assume that (s_T, Z_T) is one of the initial states of T_1 (step 16) and show that $(Z_T, L(s_T) \cap AP')$ is an initial state of T_2 (step 21) that simulates (s_T, Z_T) by definition of $\hat{H}$ (step 18). The invariant condition follows by showing that for any T_2-state s_x that simulates (s_T, Z_T) (step 23), the label of s_x has to match the label of (s_T, Z_T) (step 24).

To prove the inductive condition, let (Z_B, σ_B) be a state of T_2 that simulates (s_T, Z_T), and (s'_T, Z'_T) be a state of T_1 that is reachable in one transition from (s_T, Z_T) (step 26). Since (s'_T, Z'_T) is reachable in T_1 (step 28), by definition of $\hat{H}$, state $(Z'_T, L(s'_T) \cap AP')$ simulates (s'_T, Z'_T) (step 29). Based on definition of R_2 and $\hat{H}$, it can be shown that state $(Z'_T, L(s'_T) \cap AP')$ has to be reachable in one transition from (Z_B, σ_B) (step 34). The existence of such state $(Z'_T, L(s'_T) \cap AP')$ guarantees the satisfaction of the inductive condition.

- Proof of G4:

The proof of goal G4 can be found in figures 2.11 and 2.12. In the proof, we show

1	$\forall\, s.\ (s, \{\}) \notin \mathit{Reach}(T_1)$	
2	$\hat{H} \equiv \{((s, Z), (Z, L(s) \cap AP')) \mid (s, Z) \in \mathit{Reach}(T_1)\}$	
3	$\hat{H}((s_T, Z_T), (Z_B, \sigma_B))$	
4	$(s_T, Z_T) \in S_1$	2,3
5	$Z_B = Z_T \neq \{\} \wedge \sigma_B = L(s_T) \cap AP'$	1-3
6	$\exists\, (s', Z') \in S_1.\ R_1((s_T, Z_T), (s', Z')) \wedge (s', Z') \in \mathit{Reach}(T_1)$	2
7	(s'_T, Z'_T) $R_1((s_T, Z_T), (s'_T, Z'_T)) \wedge (s'_T, Z'_T) \in \mathit{Reach}(T_1)$	6
8	$Z'_T \neq \{\}$	1, 7
9	$\exists\, q' \in Z'_T, q \in Z_T, \sigma \subseteq L(s_T).\ \Delta(q, \sigma, q')$	7, 8
10	$\exists\, q' \in Q, q \in Z_B, \sigma \subseteq \sigma_B.\ \Delta(q, \sigma, q')$	5, 7, 9
11	$Z_B \subseteq Q \wedge \sigma_B \in \Sigma$	4, 5
12	$(Z_B, \sigma_B) \in S_2$	10, 11
13	$(s_T, Z_T) \in S_1 \wedge (Z_B, \sigma_B) \in S_2$	4, 12
14	$\hat{H} \subseteq S_1 \times S_2$	3, 13
15	$(s_T, Z_T) \in S_1$	
16	$(s_T, Z_T) \in I_1$	
17	$(s_T, Z_T) \in \mathit{Reach}(T_1)$	16
18	$\hat{H}((s_T, Z_T), (Z_T, L(s_T) \cap AP'))$	2, 17

Figure 2.9: Proof of G3

Line	Statement	Justification
19	$(Z_T, L(s_T) \cap AP') \in S_2$	14, 16, 18
20	$Z_T = \{\dot{q}\}$	15
21	$(Z_T, L(s_T) \cap AP') \in I_2$	19, 20
22	$(s_T, Z_T) \in I_1 \Longrightarrow \exists\, s_2 \in I_2.\ \hat{H}((s_T, Z_T), s_2)$	16, 18, 21
23	$\hat{H}((s_T, Z_T), s_x)$	
24	$L_1(s_T, Z_k) \cap AP_2 = L(s_T) \cap AP' = L_2(s_x)$	2, 23
25	$\forall\, s_2.\ \hat{H}((s_T, Z_T), s_2) \Longrightarrow L_1((s_T, Z_T)) \cap AP_2 = L_2(s_2)$	23, 24
26	$\hat{H}((s_T, Z_T), (Z_B, \sigma_B)) \wedge R_1((s_T, Z_T), (s'_T, Z'_T))$	
27	$(s_T, Z_T) \in \mathit{Reach}(T_1)$	2, 26
28	$(s'_T, Z'_T) \in \mathit{Reach}(T_1)$	26, 27
29	$\hat{H}((s'_T, Z'_T), (Z'_T, L(s'_T) \cap AP'))$	2,28
30	$Z_B = Z_T \wedge \sigma_B = L(s_T) \cap AP'$	2,26
31	$Z'_T = \{q' \mid \exists\, q \in Z_T, \sigma \subseteq L_1(s_T).\ \Delta(q, \sigma, q')\}$	26
32	$Z'_T = \{q' \mid \exists\, q \in Z_B, \sigma \subseteq \sigma_B.\ \Delta(q, \sigma, q')\}$	30,31
33	$(Z_B, \sigma_B) \in S_2 \wedge (Z'_T, L(s'_T) \cap AP') \in S_2$	26, 29
34	$R_2((Z_B, \sigma_B), (Z'_T, L(s'_T) \cap AP'))$	32, 33
35	$\forall\, s_2, s'_1.\ \hat{H}((s_T, Z_T), s_2) \wedge R_1((s_T, Z_T), s'_1)$ $\Longrightarrow \exists\, s'_2.\ \hat{H}(s'_1, s'_2) \wedge R_2(s_2, s'_2)$	26, 29, 34
36	$\hat{H} \subseteq S_1 \times S_2 \wedge \forall\, s_1.$ $s_1 \in I_1 \Longrightarrow \exists\, s_2 \in I_2.\ \hat{H}(s_1, s_2)$ $\wedge \forall\, s_2.\ \hat{H}(s_1, s_2) \Longrightarrow L_1(s_1) \cap AP_2 = L_2(s_2)$ $\wedge \forall\, s_2, s'_1.\ \hat{H}(s_1, s_2) \wedge R_1(s_1, s'_1)$ $\Longrightarrow \exists\, s'_2.\ \hat{H}(s'_1, s'_2) \wedge R_2(s_2, s'_2)$	14, 15, 22, 25, 35
37	$T_1 \preceq T_2$	36

Figure 2.10: Proof of G3 (Continued)

that for any value of s, none of the states $(s, \{\})$ can be reached in T_1 (step 23), if T_2 simulates T_1. We realize that by assuming the existence of a binary relation $\hat{H} \subseteq S_1 \times S_2$ that satisfies the initial, invariant, and inductive simulation conditions (step 1), and show that an arbitrary reachable T_1-state s_x (step 2) cannot have the empty set as its second component (step 22).

Suppose that s_x is located on a run of T_1 named s_{T_1} (step 5). Define a sequence Z_T and a word w_T such that for each i, Z_T^i equals the set of B-states associated with $s_{T_1}^i$ (step 6) and w_T^i equals the label of $s_{T_1}^i$ restricted to the atomic propositions in AP' (step 7). Next, we show by induction that for all integer values of i, every pair (Z_T^i, w_T^i) has to be a state of T_2 (step 20) and hence Z_T^i has to be non-empty by definition of S_2 (step 21) which is sufficient to prove that s_x does not have the empty set as its second component (step 22).

To prove the base case of the induction, we use the initial and invariant simulation conditions (step 1) to show that (Z_T^0, w_T^0) has to be a T_2-state that simulates $s_{T_1}^0$ (step 12). For the inductive case, we assume that (Z_T^j, w_T^j) is a T_2-state that simulates $s_{T_1}^j$ where j is an integer (step 13), and use the inductive and invariant simulation conditions (step 1) to show that (Z_T^{j+1}, w_T^{j+1}) is a T_2-state that simulates $s_{T_1}^{j+1}$ (step 19).

2.4 Concluding Remarks

Our SSLTL verification strategy is meant to be implemented on top of an SMT solver (such as CVC3 [3] or Z3 [47]) or an invariant checker (such as UCLID [6]). In this case, function *Check* can be viewed as a call to the SMT solver or the invariant checker. Also, the operation of function *Unfold* can be realized using a symbolic simulator such as the one built in the UCLID tool.

The invariant e taken as input by function *Verify* can be determined through an iterative process. As it is typical for induction, e is initially given a weak value (true for instance) and gradually strengthened based on information from the counterexamples. The final

1	$\hat{H} \subseteq S_1 \times S_2 \wedge \forall\, s_1.$ $s_1 \in I_1 \Longrightarrow \exists\, s_2 \in I_2.\ \hat{H}(s_1, s_2)$ $\wedge \forall\, s_2.\ \hat{H}(s_1, s_2) \Longrightarrow L_1(s_1) \cap AP_2 = L_2(s_2)$ $\wedge \forall\, s_2, s_1'.\ \hat{H}(s_1, s_2) \wedge R_1(s_1, s_1')$ $\Longrightarrow \exists\, s_2'.\ \hat{H}(s_1', s_2') \wedge R_2(s_2, s_2')$	
2	$s_x \in \mathit{Reach}(T_1)$	
3	$\exists\, \pi \in \mathit{Runs}(T_1), i \in \mathbb{N}.\ s_x = \pi^i$	2
4	$\exists\, \pi.\ \pi^0 \in I_1 \wedge \forall\, l \in \mathbb{N}.\ R_1(\pi^l, \pi^{l+1})$ $\wedge \exists\, i \in \mathbb{N}.\ s_x = \pi^i$	3
5	s_{T_1} $s_{T_1}^0 \in I_1 \wedge \forall\, l \in \mathbb{N}.\ R_1(s_{T_1}^l, s_{T_1}^{l+1})$ $\wedge \exists\, i \in \mathbb{N}.\ s_x = s_{T_1}^i$	4
6	$Z_T \equiv \lambda\, i \in \mathbb{N}.\ (\lambda\, (s, Z).\ Z)\ s_{T_1}^i$	
7	$w_T \equiv \lambda\, i \in \mathbb{N}.\ L_1(s_{T_1}^i) \cap AP'$	
8	$Z_T^0 = \{\dot{q}\}$	5, 6
9	$\exists\, s_2 \in I_2.\ \hat{H}(s_{T_1}^0, s_2)$	1, 5
10	$s_{T_2}^0$ $s_{T_2}^0 \in I_2 \wedge \hat{H}(s_{T_1}^0, s_{T_2}^0)$	9
11	$s_{T_2}^0 = (Z_T^0, w_T^0)$	1, 7, 8, 10
12	$(Z_T^0, w_T^0) \in S_2 \wedge \hat{H}(s_{T_1}^0, (Z_T^0, w_T^0))$	10, 11
13	$j \in \mathbb{N} \wedge (Z_T^j, w_T^j) \in S_2$ $\wedge \hat{H}(s_{T_1}^j, (Z_T^j, w_T^j))$	
14	$R_1(s_{T_1}^j, s_{T_1}^{j+1})$	5, 13

Figure 2.11: Proof of G4

15	$\exists\, s'_2.\ \hat{H}(s_{T_1}^{j+1}, s'_2) \wedge R_2(s_{T_2}^{j}, s'_2)$	1, 13, 14
16	$s_{T_2}^{j+1}$ $\hat{H}(s_{T_1}^{j+1}, s_{T_2}^{j+1}) \wedge R_2(s_{T_2}^{j}, s_{T_2}^{j+1})$	15
17	$Z_k^{j+1} = \{q' \mid \exists\, q \in Z_T^j, \sigma \subseteq w_T^j.$ $\Delta(q, \sigma, q')\}$	5-7
18	$s_{T_2}^{j+1} = (Z_T^{j+1}, w_T^{j+1})$	6, 7,16, 17
19	$(Z_T^{j+1}, w_T^{j+1}) \in S_2$ $\wedge\ \hat{H}(s_{T_1}^{j+1}, (Z_T^{j+1}, w_T^{j+1}))$	16, 18
20	$\forall\, i \in \mathbb{N}.\ (Z_T^i, w_T^i) \in S_2$	12, 13, 19
21	$\forall\, i \in \mathbb{N}.\ Z_T^i \neq \{\}$	9, 10, 20
22	$\forall\, s.\ (s, \{\}) \neq s_x$	4, 5, 21
23	$\forall\, s.\ (s, \{\}) \notin \mathit{Reach}(T_1)$	2,22

Figure 2.12: Proof of G4 (Continued)

value of e is also dependent on the size of the induction window k. Larger values of k likely lead to relatively weaker final values for e.

2.5 Summary

SSLTL is the largest fragment of LTL that is safe by syntax. Our approach for verifying an SSLTL property about a model is to compile the property into an automaton and an invariant. The automaton is combined together with the model to form the augmented model. The invariant is checked against the augmented model using k-step induction. The invariant can be manually strengthened to keep induction within the reachable state space. Theorem 2.1 shows that our approach is sound and complete. A tool implementing our approach is introduced in section 4.1.

Chapter 3

Inter-Instruction Dependency Correctness Criteria

This chapter explains our strategy for verifying whether a pipelined microprocessor preserves data and control dependencies among instructions. Section 3.1 provides some background information about pipelining in microprocessors. It also covers the related research in the area of formal verification of microprocessors. Section 3.2 presents a simple pipeline example (called *SimPipe*) that we use for illustration throughout the chapter. Section 3.3 explains how pipelines are described based on the behavior of their parcels (or instructions). Most of the concepts presented in section 3.3 are revisited and formalized in section 3.4.

Different aspects of the pipeline-correctness based on the behavior of parcels are expressed in section 3.5. Section 3.6 introduces the criteria based on which we determine whether inter-parcel dependencies are correctly handled. In section 3.7, the criteria are decomposed into smaller properties to reduce verification complexity. The soundness of the decomposition is shown in section 3.8. The chapter is summarized in section 3.9.

3.1 Background and Related Work

Subsection 3.1.1 describes pipelining in the context of microprocessors. It explains the potential conflicts (*i.e.*, hazards) that may arise in a pipelined system, and illustrates some of the techniques to avoid those conflicts in pipelined microprocessors. Subsection 3.1.2 covers the related research conducted in the area of formal verification of microprocessors.

3.1.1 Pipelining in Microprocessors

Pipelining is an implementation technique used in digital systems (especially microprocessors) for enhancing performance [33, 22]. A pipelined system, also referred to as a *pipeline*, is analogous to an *assembly line*: items are processed in an overlapped manner, and they have to go through many steps each of which contributes something towards the final product. The items (to be) processed by a pipeline are referred to as pipeline parcels, or just as *parcels*. The steps in a pipeline are called *stages*. Different pipeline stages process different parcels in parallel. The state of the pipeline (*e.g.*, parcels progress) is recorded within a set of *storage elements* (also known as the *physical variables*). Among storage elements, the ones used in passing parcels from one stage to another are widely known as *pipeline registers*.

Figure 3.1 shows a simple 5-stage pipeline. The life cycle of a parcel starts at stage S_1 and ends at stage S_5. During its life cycle, a parcel, described as *in-flight*, flows through the pipeline and interacts with (*i.e.*, *reads* from or *writes* to) the storage elements. A parcel proceeds to a stage by moving to the pipeline register at its input, *e.g.*, a parcel proceeds to S_3 by moving to E_{23}. A parcel may skip some stages (*e.g.*, S_2) and may repeat others (*e.g.*, S_4).

The major motivation for pipelining is to improve the performance of a system, as measured by throughput*, without a significant increase in the system's area [55]. This potential improvement in throughput is due to the overlapped processing of parcels as opposed to the sequential processing of parcels in non-pipelined systems.

*The *throughput* is defined as the average number of items (parcels) processed per unit of time

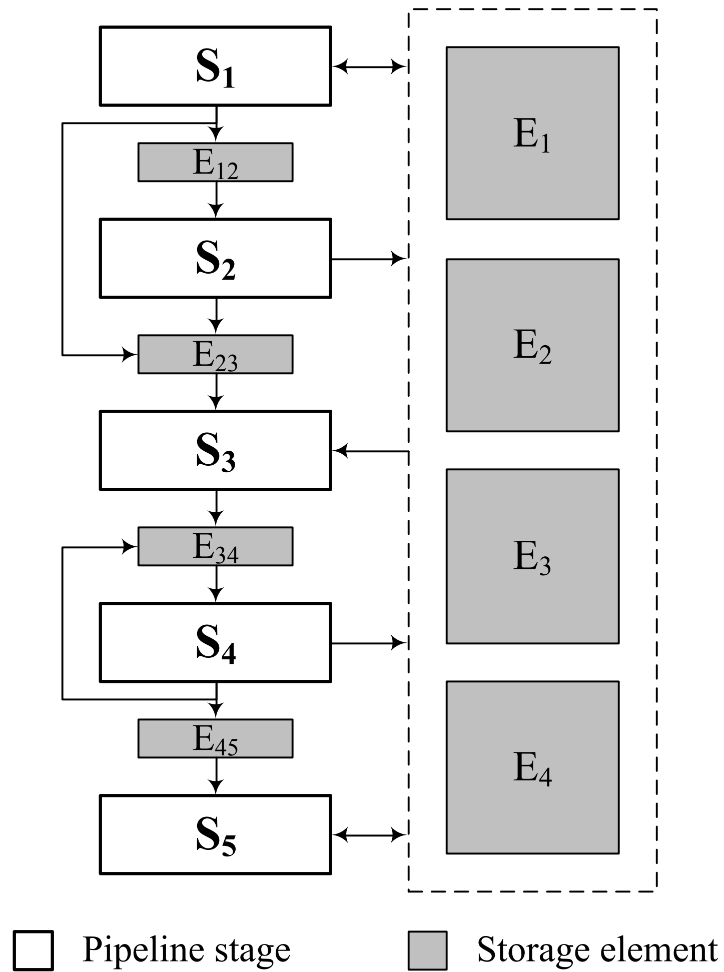

Figure 3.1: A 5-stage pipeline

The performance improvement achieved using a pipelined system over a non-pipelined one comes at expense of a significant increase in design complexity. Figure 3.1 reflects some of the structural complexities that may be present in non-linear pipelines. For instance, some of the stages (*e.g.*, S_1) have multiple successors while others (*e.g.*, S_4) have multiple predecessors. This sort of structural complexities must be considered in designing the pipeline in order to avoid livelock and deadlock.

The structural aspect is just one of the hurdles encountered when designing a pipeline. Generally, the hurdles every pipeline-designer has to deal with are widely known as *pipeline hazards*. Although they are generic to pipelined systems, pipeline hazards are easier explained in the context of pipelined microprocessors where *instructions are treated as parcels*. Conventionally, pipeline hazards are classified into three categories [22, 56]:

1. *Structural Hazards*: are resource conflicts that may happen when some of the in-flight instructions try to access shared resources simultaneously. For instance in figure 3.1, a structural hazard arises from the fact that the two instructions at stages S_1 and S_2 may try to proceed to stage S_3 simultaneously.

2. *Control Hazards*: are changes in the sequential flow of instructions caused by either flow-control instructions (*e.g.*, branches and subroutine calls), exceptions, or interrupts. In each of these three cases, the fetching sequence may be disrupted and some of the in-flight instructions may be discarded. For instance when an instruction raises an exception, based on the severity of such exception, the processor may need to cancel some of the in-flight instructions before it resteers the fetch to the exception handler.

3. *Data Hazards*: are data dependencies that may exist among the in-flight instructions. These dependencies impact the order in which instructions are processed by the pipeline. There are three types of data dependencies:

   (a) *Read-After-Write* (RAW): happens when an instruction (*consuming instruction*) depends on the result of an earlier instruction (*producing instruction*). For instance in figure 3.2, there is a RAW dependency between instructions p_1

and p_2 with respect to register r_3. RAW is considered a *true dependency* because it reflects the flow of data among instructions.

(b) *Write-After-Write* (WAW): happens when two instructions write their results to the same location. This is called *output dependency*. In figure 3.2, p_1 and p_3 have an output dependency with respect to r_3.

(c) *Write-After-Read* (WAR): happens when an instruction writes to a location used as a source operand by an earlier instruction. For instance, figure 3.2 shows a WAR dependency, also known as *antidependency*, between instructions p_2 and p_3 with respect to register r_3.

Unlike RAW, neither WAW nor WAR affects the data flow. In fact, both WAW and WAR dependencies, referred to as *name dependencies*, can be removed by a technique that combines *renaming* with eager *forwarding*.

$$\begin{array}{llll} p_1: & r_3 & \longleftarrow & r_1 * \#7 \\ p_2: & r_2 & \longleftarrow & r_3 - r_2 \\ p_3: & r_3 & \longleftarrow & r_1 + \#3 \end{array}$$

Figure 3.2: Sequence of instructions

If they are not handled elegantly, pipeline hazards may result in frequent *stalls*, *i.e.*, preventing the next instruction in the instructions stream from being executed during its designated clock cycle [22], which obviously has a negative impact on the performance. The risk of performance degradation pushes designers to employ aggressive optimizations for dealing with pipeline hazards. This is the reason why pipeline hazards are potential sources of bugs during the design phase. If pipeline hazards are not handled correctly, they cause so-called *pipeline conflicts*. Both a pipeline deadlock and a premature read (*i.e.*, reading a stale value) for a source operand of an in-flight instruction are examples of pipeline conflicts.

The research outlined in this thesis is closely related to control and data hazards. In fact, when we refer to *instruction (parcel) dependencies*, we mean both types of hazards:

control and data. Structural hazards are rarely referred to in the rest of this thesis since we deal with term-level models of microarchitectural algorithms that tend to abstract away low-level details which may cause structural conflicts. Our goal for what remains in this section is to shed light on some of the microprocessor optimizations dealing with control and data hazards.

As mentioned before, branches, exceptions, and interrupts are the main causes of control hazards. For control hazards, we focus only on aspects related to the different variants of branch instructions. Exceptions are similar to branches in the sense that every instruction that may raise an exception can be considered a branch. In other words, exceptions can be modeled by branches. Interrupts are different from exceptions and branches in the sense that they occur non-deterministically. Hence, they can't be modeled as branches.

Unlike other instructions, a branch instruction has to be executed before knowing with certainty the location of the following instruction. There are many schemes for mitigating the effects of branches:

1. Pipeline stall cycles: once a branch is encountered, the fetch is suspended until the branch is executed. Obviously, this could lead to a significant loss in performance gained through pipelining.

2. Branch delay slots: the compiler fills the slots sequentially following to the branch with some instructions that are independent of the outcome of the branch. This scheme does not scale well with deeper pipelines where the number of delay slots gets larger to the extent that they cannot be filled in with useful instructions at compilation time.

3. Branch prediction: experimental results show that branch instructions exhibit quite predictable behavior patterns [55]. This scheme makes use of these patterns in identifying (through *prediction*) the instruction that follows the branch. Once identified, that instruction is fetched and executed *speculatively*. The branch prediction is *validated* when the branch execution is complete. If a misprediction is detected, the pipeline has to go through three steps for *recovering*:

   (a) the state of the pipeline at the branch-fetching time is *restored*.

(b) the instructions in the the branch shadow are *discarded*.

(c) the fetch is *resteered* towards the correct location revealed after branch execution.

The performance-loss due to misprediction (*branch penalty*) is a major concern in this scheme. Frequent mispredictions may have a devastating effect on performance. Many innovative techniques are employed in order to achieve a high *prediction accuracy*. The prediction is made for both the *branch target* and the *branch outcome* (*e.g.*, taken or not- taken). Predicting the branch target is typically done through a lookup in the so-called Branch Target Buffer (BTB), which can be viewed as a cache for branch targets tagged by branch addresses. Predicting the branch outcome can be either done statically or dynamically. Predicting the branch outcome as *always not-taken*, is an example of the static techniques. Dynamic techniques rely on keeping the *history* of branch outcome and using it in making the prediction. In the simplest case, the history is kept using a two-bit saturation counter and updated based on the actual branch outcome. More sophisticated techniques, such as two-level adaptive branch prediction [55], use an additional shift register to adapt to changing dynamic branching context. With this kind of techniques, the prediction accuracy exceeds 95%.

Several techniques, on both the software and hardware levels, are used for resolving data hazards. In the *software techniques*, the compiler is responsible for scheduling instructions in a way that preserves data dependencies while providing efficient utilization of the resources. Such *static-scheduling* techniques were commonly used in many processors (*e.g.*, the MIPS family [29]) during the 1980s. The main disadvantage of static scheduling is that the machine code generated by compilers lacks *portability*. In fact, a new implementation of a processor may require a recompilation of the existing programs. This is needed in order to provide efficient instruction scheduling that makes use of the optimizations introduced in the new implementation.

A variety of *hardware techniques* can be used for resolving different types of data hazards. Some of them are described below:

1. *Interlocking*: is a safe and simple way for resolving all types of data hazards. Additional hardware circuitry detects data dependencies (both true and name dependencies) between instructions. Once a data dependency between two instructions is detected, pipeline interlocking circuitry stalls the execution of the dependent instruction until the other instruction has produced its result. Obviously, excessive pipeline stalling is the main drawback of this technique since it considerably decreases the pipeline performance.

2. *Forwarding*: also known as *bypassing*, is a method for handling RAW data hazards. Extra hardware, called *forwarding logic* or *bypass path*, feeds the result of the producer instruction back to the front-end of the pipeline (where the source operands are read) in order to be consumed by the dependent instruction. The forwarding is done as soon as the producer's result is output by the execution units. This forwarding mechanism minimizes the time during which the pipeline stalls the execution of the consumer instruction. With deeply-pipelined and/or superscalar machines, an efficient implementation of this mechanism is very expensive, because many bypass paths and extra multiplexers have to be introduced.

3. *Dynamic Scheduling*: is an approach by which a hardware circuit, typically referred to as a *scheduler*, rearranges instructions at execution time to reduce pipeline stalls while preserving inter-instruction dependencies. Such out-of-order execution is the main characteristic distinguishing dynamic scheduling techniques. Dynamic scheduling simplifies the compiler design and solves the code-portability problem associated with static scheduling techniques. It also handles many situations where data hazards are unknown at compile time. Unfortunately, these benefits come at the expense of a significant increase in hardware complexity.

4. *Register Renaming*: is a technique for removing name dependencies (WAW and WAR) between instructions. In this technique, the architectural registers are implemented using a larger number of physical registers. The key idea to eliminate name dependencies is to avoid using the same physical register as a destination for more than one of the in-flight instructions. Each new instruction is *assigned* a unique physical register (as a destination) and the operands of the following instructions are

renamed accordingly.

5. *Value Prediction*: is a *speculative* technique for exploiting the *value locality*, *i.e.*, likelihood of recurrence of values previously seen by the instructions [39]. The technique allows instructions to proceed to execution with predicted values for their operands before the actual values are computed. This aggressive approach has the potential to push performance beyond the *data-flow limit* imposed by the true dependencies among instructions. In concept, value prediction shares some similarities with branch prediction. For instance, the predicted value has to be *validated* later on and a *recovery* sequence has to be triggered upon detecting a misprediction. However with value prediction, a complete value for the source operand(s) (*e.g.*, a 64-bit integer), as opposed to a one-bit branch outcome, has to be predicted. Therefore, value predictors (with acceptable accuracies) are much more complicated than history-based branch predictors. This is a major limitation on the applicability of value prediction.

3.1.2 Formal Verification of Microprocessors

This section discusses related work on formal verification of pipelined microprocessors. One of the main challenges in verifying pipelined processors is deciding how to relate the implementation states to the corresponding specification states. In order to justify the previous statement, let us assume that the specification is a non-pipelined processor implementing the *Instruction Set Architecture* (ISA). Assume further that the specification is able to fetch and completely execute an instruction in a single step. This means that the states visited by the specification during the execution of a program directly correspond to the boundaries between program instructions. On the other hand, the implementation processes multiple instructions in parallel and overlaps each step. Therefore, the states visited by the implementation when it executes a program are not necessarily in a direct match with the instruction boundaries. This is the reason why the implementation states cannot be directly compared to the specification states. Consequently, an *abstraction function* is needed to transform the implementation states into states that are comparable to the corresponding specification states.

Burch and Dill [8] present an automatic approach for verifying pipelined processors. The basic idea behind their approach is to push the implementation from its current state to a *flushed state*, a state where there are no in-flight instructions; a flushed state can then be compared to a specification state. For flushing the pipeline, they use an abstraction function that carries out two operations: stalling the pipeline front-end (*i.e.*, no new instructions are allowed to enter the pipeline) and proceeding with normal execution until all the in-flight instructions complete their execution. Unfortunately, the computational complexity of flushing realistic pipelines, especially for those supporting out-of-order execution, is unmanageable.

The original flushing approach does not handle any of the *liveness* aspects of the pipeline (*e.g.*, freedom of deadlocks). Velev [64] extends the flushing approach to handle liveness under certain modeling restrictions. The idea is to simulate the pipeline for some fixed number of steps, n, known to be enough for making a progress (*e.g.*, fetching an instruction). Knowledge about the implementation of the pipeline is needed in this case to determine n.

Verification scalability can be significantly improved by decomposing the verification task into smaller subtasks. Hosabettu, Srivas and Gopalakrishnan [25] devise their abstraction function as a composition of a set of *completion functions*, one per each in-flight instruction. A completion function mimics the desired effect (on observables) of completing an instruction. The completion functions are called in the order by which instructions enter the pipeline (*i.e.*, program order). In case of out-of-order execution, program order is extracted from the reorder buffer [26]. Calling completion functions in program order has an effect (on observables) that is similar to that of flushing the pipeline. Devising the abstraction function as a composition of multiple completion functions allows the use of induction over pipeline stages in comparing implementation states against specification states.

In both flushing and completion functions techniques, correctness requires comparing implementation states (after abstraction) to specification states each time the implementation takes a step; this type of correctness is called *single-step* correctness. On the other hand, *multi-step* correctness requires comparing implementation states against specification states at certain points during the implementation run. For instance, Sawada and Hunt [53] carry out the comparison whenever the implementation visits a state that hap-

pens to be a flushed state; in this case an abstraction function is not needed. Aagaard, Day and Luo [2] prove that multi-step correctness is equivalent to single-step correctness under certain conditions.

Manolois [41] expresses the correctness of pipelined microprocessors in terms of a *Well-founded Equivalence Bisimulation* (WEB). Verification is done by proving a WEB-refinement theorem that guarantees that the pipeline has exactly the same infinite executions as the specification (ISA), up to stuttering. The theorem specifies the liveness of the pipeline in terms of a function (called *rank function*) that maps the pipeline states to some ordered values which can be used to measure progress (*e.g.*, the number of steps needed to be taken in order to fetch a new instruction). The WEB-refinement theorem is extended to support a hierarchical form of compositional reasoning [42].

In another decomposition style, McMillan [44] uses knowledge about the pipeline behavior to manually derive a set of model checking obligations. These obligations are localized with respect to the logical sections of the pipeline. The verification relies on SMV support for compositional model checking.

Aagaard [1] introduces a correctness statement (*PipeOk*) for pipelined circuits based upon conventional pipeline hazards. The main idea behind this technique is that a pipelined implementation is correct if it correctly handles its structural, control and data hazards. Aagaard proves that *PipeOk* guarantees single-step correctness. *PipeOk* is expressed as a set of correctness obligations associated with different types of pipeline hazards. Based on PipeOk, Shehata and Aagaard [54] present a generic strategy for verifying *register renaming* techniques. They introduce a set of predicates to characterize register renaming schemes and provide a set of model-checking obligations that are sufficient to guarantee the data-hazard obligations in PipeOk.

The conventional approach to the formal verification of a microprocessor is to construct a single, monolithic, correctness criterion. The verification relies on lemmas and invariants that are defined on a case-by-case basis for each pipeline. The conventional approach looks at a state of the pipeline, which is problematic because the large number of in-flight parcels causes capacity problems in verification.

Our work provides a general definition of correctness and a general verification strat-

egy that decomposes the top-level correctness statement into simpler obligations about data/control dependencies between parcels on individual variables. Our approach saves the effort and potential mistakes of creating custom definitions of correctness and verification strategies for each pipeline.

3.2 Pipeline Example: ***SimPipe***

To illustrate the concepts introduced in this chapter, we use the three-stage pipeline *SimPipe* shown in figure 3.3. The purpose of *SimPipe* is to add up the initial contents of an unbounded memory array M. The ultimate goal is to store, in each location M_j, the summation of the initial contents of the preceding locations added to its initial value, *i.e.*, $\sum_{k=0}^{j} \dot{M}_k$ where $\dot{M}_k$ is the initial value of location M_k.

SimPipe adds up the the memory contents incrementally. It uses a counter C to keep track of the next memory location to be processed. Suppose C holds a value of j at a given state. The first stage (S_1) increments the counter C and reads the value of location M_{j+1}. Next, stage S_1 passes the new value of C (*i.e.*, $j+1$) along with the value read from location M_{j+1} to the following stage S_2 through registers C_{12} and D_{12} respectively. Stage S_1 also resets the (bubble-flag) register B_{12} to indicate that the contents of registers C_{12} and D_{12} are valid.

During the second stage (S_2), the value stored in register D_{12} (*i.e.*, the value of M_{j+1}) is added to the contents of register D_{23}. By doing so, register D_{23} would carry the summation of memory locations M_0 through M_{j+1}. Also during this stage, the values of B_{12} and C_{12} are copied to registers B_{23} and C_{23} respectively.

In the last stage (S_3), the content of D_{23} is written to the memory location whose address is the value of C_{23}. In other words, the summation of locations from M_0 to M_{j+1} gets stored to location M_{j+1}. This write operation takes place if and only if the value of B_{23} is false, *i.e.*, the values carried by C_{23} and D_{23} are valid.

In order for *SimPipe* to work correctly, three initial conditions need to be satisfied. First, registers C and D_{23} shall store a value of 0. Second, the value of D_{12} shall be equal

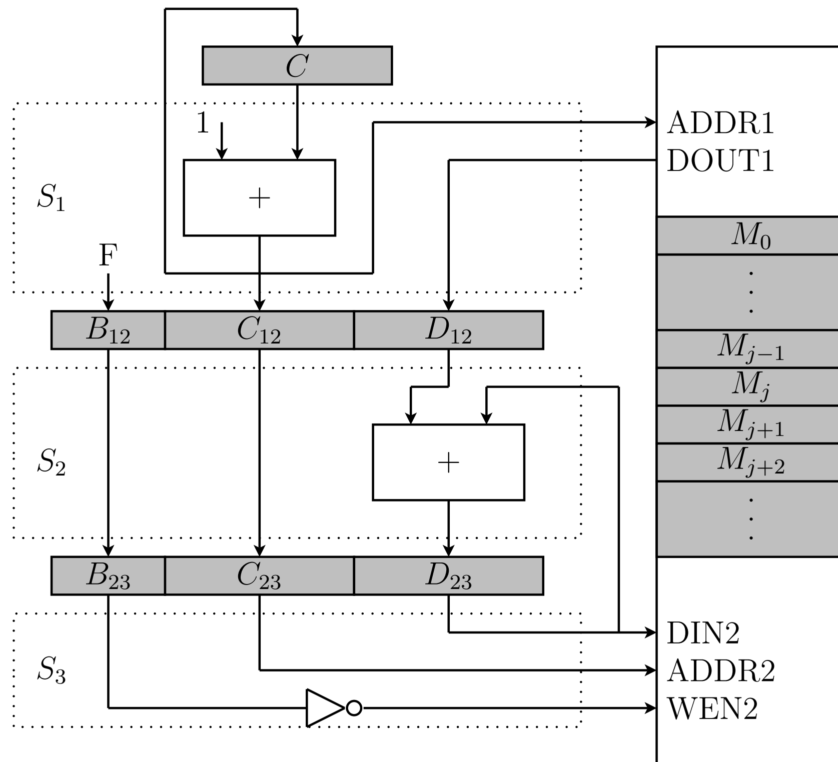

Figure 3.3: *SimPipe* pipeline

to the content of location M_0. Third, registers B_{12} and B_{23} shall store a value of true. There are no restrictions on the initial contents of registers C_{12} and C_{23}.

The functionality of *SimPipe* is best described by the non-pipelined system shown in figure 3.4. This system can be used as a *reference model*, *i.e.*, the specification machine against which *SimPipe* is compared. In the rest of this chapter, we refer to the pipelined and non-pipelined versions of *SimPipe* shown in figures 3.3 and 3.4 as the *implementation* and the *specification* of *SimPipe* respectively.

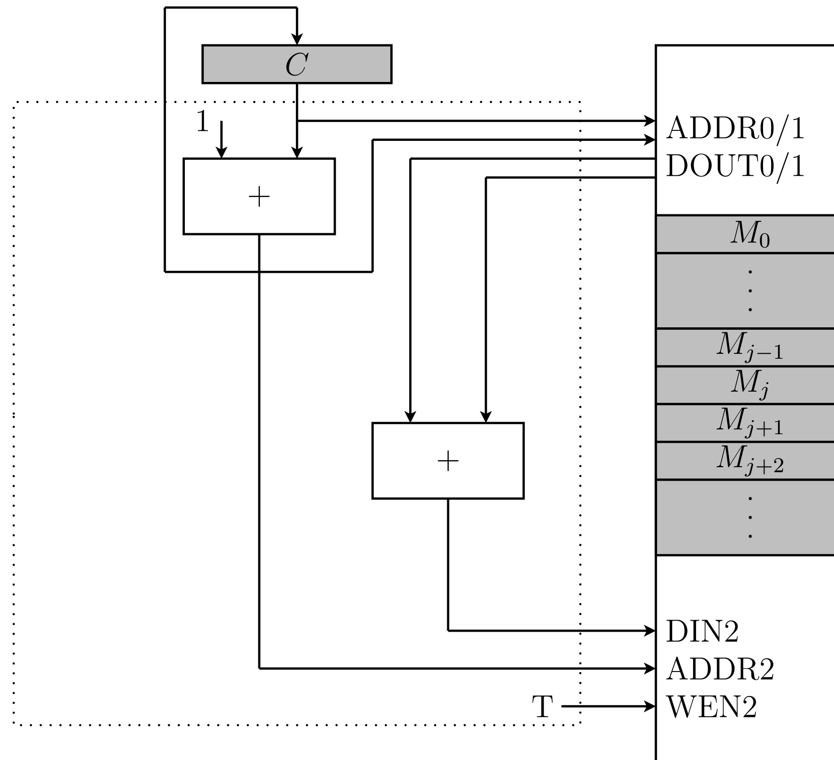

Figure 3.4: Non-pipelined specification of *SimPipe*

The specification of *SimPipe* achieves the same functionality of stages S_1, S_2, and S_3 combined together in a single step. At any given state, assuming register C holds a value of j, the system reads the values of locations M_j and M_{j+1}, adds these two values together, and writes the result back to location M_{j+1}. One major difference here is that

the specification obtains the value of the summation $\sum_{k=0}^{j} \dot{M}_k$ directly from location M_j unlike the case of the implementation where the summation value is read from register D_{23}. This is why a memory component that has two read ports is used in the specification.

3.3 Parcel-Centric View of Pipelines

The conventional analogy between a pipeline and an assembly line is made clear in subsection 3.1.1. In both systems, items (or parcels) in process go through a sequence of steps (or stages) each of which makes a contribution towards the final product (or result). More clearly in the case of a pipeline, the final result depends on the way the parcel interacts with both the state of the pipeline and the other parcels.

In this section, we focus on explaining the key concepts used in modeling both parcel-state and parcel-parcel (*i.e.*, inter-parcel) interactions. These concepts are the basic building blocks of the correctness criteria presented in the rest of the thesis. Most of the topics discussed in this section are formalized later in section 3.4.

We start by describing the different phases in the *lifetime* of a parcel. At any state, a parcel can be in one of the following four phases: *top*, *in-flight*, *retired*, and *discarded*. Before it enters the pipeline, a parcel is considered to be in the top phase. Once a parcel is fetched, its phase changes to become in-flight. The phase stays in-flight until processing the parcel is either completed or abandoned. At this point, the parcel exits the pipeline, and as a consequence, its phase changes to either retired or discarded respectively.

Example 3.1. Suppose that a parcel in the *SimPipe* implementation, the pipeline shown in figure 3.3, is identified by the natural number held by register C at the time when the parcel starts to be processed. Suppose further that C holds a value of p at the current state. This means that parcel p has just become in-flight. Parcels $p-1$ and $p-2$ are also in-flight if $p-1$ and $p-2$ are valid identifiers (*i.e.*, natural numbers). These two identifiers are valid if and only if B_{12} and B_{23} store false values, respectively. All parcels with identifiers greater than p have not entered the pipeline yet. Therefore, these parcels are in the top phase. On the other hand, all the parcels identified with natural numbers

less than $p - 2$ are no longer in-flight. Hence, these parcels are in the retired phase since no parcels are discarded in *SimPipe*.

The state of any pipeline is stored in elements called the *physical variables*. The physical variables should be distinguished from those variables that are mentioned in the specifications of the pipeline (*e.g.*, Instruction Set Architecture). We refer to the latter as the *architectural variables*. The architectural variables are not actual storage elements in the pipeline and hence they are not part of the pipeline state. The state of the pipeline is exclusively held by the physical variables since they are the actual storage elements of the pipeline.

The state of the pipeline is interpreted using an *address map* that relates the physical variables to the architectural variables. At any given state, each physical variable is mapped to *at most one* architectural variable, and for each architectural variable there exists *at least one* physical variable that is mapped to that architectural variable.

The mapping relationship between a physical variable and an architectural variable can be either static or dynamic. For instance, an architectural register can be represented in the processor implementation by two physical variables: an entry in the (physical) register file and a bypass register. In the first case, the register-file entry is dedicated to the architectural register and hence the mapping is static. In the second case, the bypass register may, over time, carry values that belong to multiple architectural registers and hence the mapping in this case is dynamic.

Example 3.2. The physical variables of the *SimPipe* implementation are C, B_{12}, C_{12}, D_{12}, B_{23}, C_{23}, D_{23}, and M_j for all $j \in \mathbb{N}$. Based on the specification of *SimPipe*, the architectural variables are C and M_j for all $j \in \mathbb{N}$. The physical variable C is *statically* mapped to the corresponding architectural variable C. On the other hand, D_{23} gets *dynamically* mapped over time to different locations in the architectural memory. The location represented by D_{23} at a given state depends on the contents of registers B_{23} and C_{23}. If the value of B_{23} is false (*i.e.*, not a bubble), then D_{23} is mapped to the location whose address is held by C_{23}, because this is where the value in D_{23} will be stored. Otherwise, D_{23} does not carry valid data, and consequently, it does not represent any locations in the architectural memory.

The specifications of the pipeline determine which architectural variables a parcel should read from and/or write to. We refer to the architectural variables that need to be read by a parcel (according to the specifications) as the *sources* of the parcel. Similarly, the architectural variables to which a parcel should write (according to the specifications) are the *destinations* of the parcel.

Parcels interact with the state of the pipeline during the in-flight phase by reading from and/or writing to the physical variables that are mapped to their sources and/or destinations respectively. This parcel-state interaction may be *speculative*. A parcel may write a speculative value to a physical variable v_1 that is mapped to one of its destinations v_a. If the parcel detects that the written value is incorrect, the parcel signals mispredict and takes steps towards recovery. To recover from a misprediction, the parcel can either make a corrective write or go to the discard phase. The corrective write can be made to any physical variable v_2, which may or may not be v_1, as long as it is mapped to v_a. The variable v_1 becomes not mapped to v_a until its contents are corrected.

A parcel can also make an arbitrary number of speculative reads from physical variables that are mapped to its sources. Similar to the write case, a parcel can read any of its sources multiple times using different physical variables that are mapped to that source. The last read by a parcel from any physical variable that is mapped to a source v_a is called the *final read* from v_a. All but the final reads are considered speculative and therefore do not affect the final results of the parcel.

The definition of the dual concept (final writes) is slightly different. A *final write* made by a parcel to a destination v_a is the latest write *of a new value* to any physical variable that is mapped to v_a. The major difference here is that the final write may take place before the last write. In other words, after a parcel makes its final write to a detention v_a, the parcel can copy that value to other physical variables that are mapped to v_a. Allowing this behavior is important for modeling forwarding techniques in microprocessors. Similar to the case of the reads, all but the final writes are considered speculative and hence should not have any impact on the final "architectural" state of the pipeline†. The *architectural state* of a pipeline is captured by the subset of physical variables that are mapped to

† *Final state* here means any state that can be reached from the current state by flushing the pipeline.

architectural variables.

Both control and data flow among parcels in a pipeline have to be consistent with the way parcels are ordered. In the context of microprocessors, this order takes the form of a program that specifies both control and data dependencies among instructions (or parcels). Inter-parcel dependencies manifest themselves in the way parcels read from and write to the physical variables.

Two situations need to be addressed in order to handle inter-parcel dependencies correctly. The first situation is when a parcel p_2 has a source v_a that is a destination of an *older* parcel p_1 (*i.e.*, p_1 comes in order before p_2), and v_a is not a destination of any parcel that comes in between p_1 and p_2. In this case, we refer to p_1 and p_2 as *the producer* and *the consumer* respectively, and we briefly describe this situation by saying that there is a *direct dependency* from p_1 to p_2.

The second situation is about a consumer that has no producer. This happens when a parcel p has a source v_a that is not the destination of any older parcel. We describe this situation by saying there is *no dependency* from any parcel to p. In section 3.6, we present two minimally-restrictive rules to guarantee that inter-parcel dependencies are preserved in both the direct-dependency and no-dependency situations.

3.4 Parcel-Based Instrumentation of Pipelines

Specifying criteria to judge the correctness of the implementation is a key step in formal verification. In the case where the focus is on verifying whether a pipeline preserves dependencies between parcels, devising these criteria requires a mechanism for identifying parcels and marking certain events that take place during their lifetime. In this section, we show how a pipeline can be systematically instrumented to track individual parcels and monitor their interaction with each other as well as with the pipeline state variables (*i.e.*, physical variables). We use this instrumentation technique in section 3.6 to specify properties to guarantee that parcel-to-parcel communication is done properly. Other aspects of pipeline-correctness can be expressed using this instrumentation technique as explained in section 3.5.

We start by presenting a mathematical model for pipelines. A *pipeline* $\mathcal{X}$ is defined as a six-tuple $\langle V, V_a, Q, \dot{Q}, T, \mathrm{AM}\rangle$ where:

- V is the set of *physical variables*.
- V_a is the set of *architectural variables*.
- Q is the set of *states*. Each state is an *environment* (over V) that maps each variable in V to a *value*. The value of a variable $v \in V$ at a state $q \in Q$ is denoted by $q.v$.
- $\dot{Q} \subseteq Q$ is the set of *initial states*.
- $T \subseteq Q \times Q$ is the *transition relation*.
- $\mathrm{AM} \subseteq V \times V_a \times Q$ is the *address map* predicate. Notationally, $\mathrm{AM}_v^{v_a}\ q$ means that a physical variable v is mapped to an architectural variable v_a at a state q. It is possible that v does not represent any architectural variables at a state q, *i.e.*, $\neg\mathrm{AM}_v^{v'_a}\ q$ for all $v'_a \in V_a$.

Example 3.3. The address map predicate AM for the *SimPipe* implementation can be formally defined by pattern-matching one of the following cases:

$$\begin{aligned}
\mathrm{AM}_C^C\ q &\equiv \texttt{True}\\
\mathrm{AM}_{M_j}^{M_k}\ q &\equiv (j = k)\\
\mathrm{AM}_{D_{23}}^{M_k}\ q &\equiv \neg q.B_{23} \wedge (k = q.C_{23})\\
\mathrm{AM}_{-}^{-}\ q &\equiv \texttt{False}
\end{aligned}$$

where the last case matches all the patterns that are not matched by the first three cases. Since D_{12}, C_{12}, and C_{23} are not used in exchanging either data or control information among parcels, none of these physical variables is mapped to an architectural variable.

A *run* of the pipeline $\mathcal{X}$ is an infinite sequence of states $\sigma = \ll \sigma^0\sigma^1\sigma^2 \ldots \gg$ where $\sigma^0 \in \dot{Q}$ and $T(\sigma^i, \sigma^{i+1})$ for all $i \in \mathbb{N}$. The set of runs of $\mathcal{X}$ is denoted as $\mathit{Runs}(\mathcal{X})$.

To be able to track parcels, we augment the pipeline with a mechanism for identifying parcels and a set of predicates that captures relevant events in the lifetime of those parcels. More specifically, the pipeline $\mathcal{X}$ is augmented by adding the following three components:

1. *Parcel identifiers*: an ordered set $\langle P, \prec \rangle$ where P is an infinite set of parcel identifiers and $\prec \; \subseteq P \times P$ is a total order over these identifiers.

   Example 3.4. The ordered set $\langle P, \prec \rangle$ used to identify parcels in *SimPipe* can be defined to be $\langle \mathbb{N}, < \rangle$

2. *Phase predicates*: used to probe the phase of a parcel at any given state. Four predicates are used for that purpose:

   (a) Top predicate ($\text{Top} \subseteq P \times Q$): holds for parcels that have not entered the pipeline yet.

   (b) In-flight predicate ($\text{Infl} \subseteq P \times Q$): holds for parcels that are being processed by the pipeline.

   (c) Discarded predicate ($\text{Dis} \subseteq P \times Q$): holds for parcels that have been discarded and no longer processed by the pipeline.

   (d) Retired predicate ($\text{Ret} \subseteq P \times Q$): holds for parcels that have been completely processed and exited the pipeline.

   Example 3.5. The phase predicates in the *SimPipe* implementation can be defined as follows:

   $$\begin{aligned}
   \text{Top}\ p\ q &\equiv p > q.C \\
   \text{Infl}\ p\ q &\equiv p \leq q.C \wedge p \geq q.C - 2 \\
   \text{Ret}\ p\ q &\equiv p < q.C - 2 \\
   \text{Dis}\ p\ q &\equiv \texttt{False}
   \end{aligned}$$

3. *Interaction predicates*: used to capture events in which parcels interact with the state of the pipeline. Two predicates are used for that purpose (supposing $\mathcal{D}$ is the set of values that can be held by variables in V):

   (a) Read predicate ($\text{Rd} \subseteq V \times V_a \times \mathcal{D} \times P \times Q$): holds when a parcel reads from a variable. Notationally, $\text{Rd}_v^{v_a}\ d\ p\ q$ means that a parcel p reads a data value d from a physical variable v that is mapped to an architectural variable v_a. Notice that the mapping relationship between v and v_a should hold at the time when

the read takes place, which can be formally captured as follows:

$\forall\, \sigma, v, v_a, p, d, j.\ \sigma \in \mathit{Runs}(\mathcal{X}).$

$\qquad \mathrm{Rd}_v^{v_a}\ d\ p\ \sigma^j \Longrightarrow \mathrm{AM}_v^{v_a}\ \sigma^j$

Example 3.6. The read predicates in the *SimPipe* implementation can be defined as follows:

$$\begin{aligned}
\mathrm{Rd}_C^C\ d\ p\ q &\equiv p = q.C \wedge d = q.C \wedge \mathrm{AM}_C^C\ q \\
\mathrm{Rd}_{M_j}^{M_k}\ d\ p\ q &\equiv p = q.C \wedge d = q.M_j \wedge j = q.C + 1 \wedge \mathrm{AM}_{M_j}^{M_k}\ q \\
\mathrm{Rd}_{D_{23}}^{M_k}\ d\ p\ q &\equiv \neg q.B_{12} \wedge p = q.C_{12} \wedge d = q.D_{23} \wedge \mathrm{AM}_{D_{23}}^{M_k}\ q \\
\mathrm{Rd}_{--}^{--}\ d\ p\ q &\equiv \texttt{False}
\end{aligned}$$

(b) Write predicate ($\mathrm{Wr} \subseteq V \times V_a \times \mathcal{D} \times P \times Q$): holds when a parcel writes to a variable. The write predicate (Wr) is notationally similar to the read predicate (Rd). However, when the write takes place at a state q, the mapping relationship holds in the following state q'. Also, the written value becomes available at q'. These two characteristics are described by the following formula:

$\forall\, \sigma, v, v_a, p, d, j.\ \sigma \in \mathit{Runs}(\mathcal{X}).$

$\qquad \mathrm{Wr}_v^{v_a}\ d\ p\ \sigma^j \Longrightarrow \mathrm{AM}_v^{v_a}\ \sigma^{j+1} \wedge \sigma^{j+1}.v = d$

Example 3.7. The write predicates in the *SimPipe* implementation can be defined as follows:

$$\begin{aligned}
\mathrm{Wr}_C^C\ d\ p\ q &\equiv p = q.C \wedge d = q.C + 1 \\
\mathrm{Wr}_{M_j}^{M_k}\ d\ p\ q &\equiv \neg q.B_{23} \wedge p = q.C_{23} \wedge d = q.D_{23} \wedge j = k \\
\mathrm{Wr}_{D_{23}}^{M_k}\ d\ p\ q &\equiv \neg q.B_{12} \wedge p = q.C_{12} \wedge d = q.D_{12} + q.D_{23} \wedge k = q.C_{12} \\
\mathrm{Wr}_{--}^{--}\ d\ p\ q &\equiv \texttt{False}
\end{aligned}$$

In addition to the parcel predicates used for augmenting the pipeline $\mathcal{X}$, we introduce four shortcut predicates:

1. Fetch predicate ($\mathrm{Fetch} \subseteq P \times Q$): holds when a parcel (among those in the top phase) is about to enter the pipeline. This happens right before a parcel becomes in-flight. The following formula captures the semantics of the fetch predicate:

   $\forall\, \sigma, p, j.\ \sigma \in \mathit{Runs}(\mathcal{X}).$

   $\qquad \mathrm{Fetch}\ p\ \sigma^j \Longleftrightarrow \mathrm{Top}\ p\ \sigma^j \wedge \mathrm{Infl}\ p\ \sigma^{j+1}$

2. Retire predicate (Retire $\subseteq P \times Q$): holds when an in-flight parcel is about to exit the pipeline after completion. This is immediately before that parcel becomes retired. The semantics of the retire predicate are formalized as follows:
$\forall\, \sigma, p, j.\ \sigma \in \mathit{Runs}(\mathcal{X}).$
$\quad \text{Retire}\ p\ \sigma^j \iff \text{Infl}\ p\ \sigma^j \wedge \text{Ret}\ p\ \sigma^{j+1}$

3. Final-read predicate (FRd $\subseteq V \times V_a \times \mathcal{D} \times P \times (\mathbb{N} \rightarrow Q) \times \mathbb{N}$): holds when a parcel reads a value associated with an architectural variable and afterwords the parcel makes no other reads from any physical variables mapped to that architectural variable. The final-read predicate is formalized as:
$\forall\, \sigma, v, v_a, p, d, j.\ \sigma \in \mathit{Runs}(\mathcal{X}).$
$\quad \text{FRd}_v^{v_a}\ d\ p\ \sigma\ j$
$\quad \iff$
$\quad \text{Rd}_v^{v_a}\ d\ p\ \sigma^j$
$\quad \wedge\, \forall\, v', d', k.\ k > j.\ \neg(\text{Rd}_{v'}^{v_a}\ d'\ p\ \sigma^k)$

4. Final-write predicate (FWr $\subseteq V \times V_a \times \mathcal{D} \times P \times (\mathbb{N} \rightarrow Q) \times \mathbb{N}$): holds in a state at which a parcel writes the final value of an architectural variable. Unlike the definition of the final read, following writes may happen as long as the value being written is always the same. The final-write predicate is formalized as:
$\forall\, \sigma, v, v_a, p, d, j.\ \sigma \in \mathit{Runs}(\mathcal{X}).$
$\quad \text{FWr}_v^{v_a}\ d\ p\ \sigma\ j$
$\quad \iff$
$\quad \text{Wr}_v^{v_a}\ d\ p\ \sigma^j$
$\quad \wedge\, \forall\, v', d', k.\ d' \neq d, k > j.\ \neg(\text{Wr}_{v'}^{v_a}\ d'\ p\ \sigma^k)$

Other variations of the predicates presented in this section are needed to express different aspects of correctness concisely. Instead of introducing new predicates, we overload the predicates presented above in three different ways:

1. We use a run instead of a state to mean that at least one of the states within that run satisfies the predicate. For instance, Retire $p\ \sigma$ where $\sigma \in \mathit{Runs}(\mathcal{X})$ is equivalent to $\exists\, i.$ Retire $p\ \sigma^i$.

2. We remove the state argument when the predicates are used in a context in which states are implicit. For example, in the context of linear temporal logic, we write $\mathbf{G}\ \neg\mathrm{Wr}_v^{v_a}\ p$ to mean that $\mathrm{Wr}_v^{v_a}\ p\ q$ never holds in any future state q.

3. We remove arguments other than the state as a means of existential quantification. For instance, $\mathrm{Wr}^{v_a}\ p\ q$ is equivalent to $\exists\, v', d'.\ \mathrm{Wr}_{v'}^{v_a}\ d'\ p\ q$. Using this notation, we can briefly express an anonymous write to a physical variable v at a state q as: $\mathrm{Wr}_v\ q$.

3.5 Parcel-Based Correctness of Pipelines

This section describes ongoing work in collaboration with Aagaard. Our individual work resumes in section 3.6. The collaborative work is aimed at a general, high-level, and complete definition of correctness independent of any particular verification technique. Inter-parcel correctness is a key aspect of overall correctness. As defined later in this section, inter-parcel correctness refers to the behavior of data values across potentially distant points in time in an infinite stream of computation. In section 3.6, we take the stream- and data-based definition of inter-parcel correctness and translate it into a form that is more amenable to automated verification.

The focus in this section is to express the correctness of a pipeline in terms of the behavior of its parcels. The main idea is to compare the writes made by the parcels in a pipeline (*i.e.*, implementation) against those made by the corresponding parcels in a non-pipelined reference model (*i.e.*, specification). The pipeline is said to be correct if those two sets of writes are equivalent.

Suppose that the *implementation* is a pipeline $\mathcal{I} = \langle V_i, V_a, Q_i, \dot{Q}_i, T_i, \mathrm{AM}\rangle$ that is augmented with the four components:

1. an ordered set of parcel identifiers: $\langle P_i, \prec\rangle$.

2. a set of phase predicates: $\{\mathrm{Top}, \mathrm{Infl}, \mathrm{Ret}, \mathrm{Dis}\}$.

3. a set of interaction predicates: $\{\mathrm{Rd}, \mathrm{Wr}\}$.

4. a set of shortcut predicates: $\{\text{Fetch}, \text{Retire}, \text{FRd}, \text{FWr}\}$.

By definition, the architectural variables in the implementation are those variables referenced by the specification. On that basis, we model the specification as a pipeline whose physical variables match those in V_a. In other words, let the specification be a pipeline $\mathcal{S} = \langle V_a, V_a, Q_s, \dot{Q}_s, T_s, \{(v_a, v_a, q_s) \mid v_a \in V_a \wedge q_s \in Q_s\}\rangle$. Notice that the address map predicate is defined such that each variable is mapped to itself.

In order to track the writes of parcels in the specification, $\mathcal{S}$ is augmented with two components:

1. an ordered set of parcel identifiers. To simplify the presentation, we assume without loss of generality that parcels in the specification are identified by natural numbers, *i.e.*, the ordered set of identifiers is $\langle \mathbb{N}, < \rangle$.
2. a write predicate SWr.

In the specification, parcels are neither overlapped nor speculatively processed. In each step the specification fetches a new parcel and processes it until completion. Therefore, parcels processed in a given run of the specification can be simply identified by state indices with that run. On the contrary, parcels in the implementation may be overlapped and/or discarded while being processed. Only those parcels that retire in the implementation match parcels in the specification.

Whether a parcel p_i in an implementation run σ_i *matches* a parcel p_s in a specification run σ_s, denoted $p_i \overset{\sigma_i \sigma_s}{\underset{\mathbf{PCL}}{=\!=}} p_s$, is defined by induction over p_s. In the base case, where $p_s = 0$, p_i matches p_s if and only if p_i is the first parcel to retire. In the inductive case, where $p_s > 0$, assuming parcels in the implementation retire in order, p_i matches p_s if and only if p_i is the first parcel to retire after the parcel that matches $p_s - 1$. The parcels-matching relation $\underset{\mathbf{PCL}}{=\!=}$, which is called *parcels equality*, is defined as follows:

$$
\begin{aligned}
& p_i \underset{\mathbf{PCL}}{\overset{\sigma_i\sigma_s}{=\!=}} p_s \equiv \\
& \quad \mathbf{BASE}\ (p_s = 0): \\
& \qquad \text{Retire } p_i\ \sigma_i \\
& \qquad \wedge \forall\, p_i' \prec p_i.\ \neg(\text{Retire } p_i'\ \sigma_i) \\
& \quad \mathbf{INDUCTIVE}\ (p_s > 0): \\
& \qquad \text{Retire } p_i\ \sigma_i \\
& \qquad \wedge \exists\, p_i' \prec p_i. \\
& \qquad\quad p_i' \underset{\mathbf{PCL}}{\overset{\sigma_i\sigma_s}{=\!=}} p_s - 1 \\
& \qquad\quad \wedge \forall\, p_i''.\ p_i' \prec p_i'' \prec p_i.\ \neg(\text{Retire } p_i''\ \sigma_i)
\end{aligned}
$$

The equivalence between the final writes made by p_i during σ_i and those made by p_s during σ_s is denoted as $p_i \underset{\mathbf{FWr}}{\overset{\sigma_i\sigma_s}{=\!=}} p_s$. Such an equivalence means that both p_i and p_s write to the same set of architectural variables. It also means that the values produced during the final writes of p_i are the same as those written by p_s. The relation $\underset{\mathbf{FWr}}{=\!=}$, which is called *final-writes equality*, is defined as follows:

$$
\begin{aligned}
& p_i \underset{\mathbf{FWr}}{\overset{\sigma_i\sigma_s}{=\!=}} p_s \equiv \\
& \quad \forall\, v_a. \\
& \qquad \text{Wr}^{v_a}\ p_i\ \sigma_i \Longleftrightarrow \text{SWr}^{v_a}\ p_s\ \sigma_s \\
& \qquad \wedge \forall\, v_i, j, k. \\
& \qquad\quad \text{FWr}^{v_a}_{v_i}\ p_i\ \ \sigma_i\ j \\
& \qquad\quad \wedge\ \text{SWr}^{v_a}\ p_s\ \sigma_s^k \\
& \qquad\quad \Longrightarrow \\
& \qquad\quad \sigma_i^{j+1}.v_i = \sigma_s^{k+1}.v_a
\end{aligned}
$$

Given the parcels equality and the final-writes equality defined above, the correctness is stated by saying that: *the final writes of each parcel that retires in the implementation $\mathcal{I}$ shall be equivalent to the writes of the matching parcel in the specification $\mathcal{S}$.* We refer to this correctness statement as the *final-writes containment.* The final-writes containment is formally expressed as follows:

$$
\begin{aligned}
& \forall\, \sigma_i, p_i, p_s.\ \sigma_i \in \mathit{Runs}(\mathcal{I}).\ \exists\, \sigma_s \in \mathit{Runs}(\mathcal{S}). \\
& \quad p_i \underset{\mathbf{PCL}}{\overset{\sigma_i\sigma_s}{=\!=}} p_s \Longrightarrow \sigma_i \underset{\mathbf{FWr}}{\overset{p_i p_s}{=\!=}} \sigma_s
\end{aligned}
$$

The final-writes containment can be decomposed into two criteria. The first of which guarantees that a parcel in the implementation would behave correctly in isolation from

other parcels. We refer to this aspect as the *intra-parcel correctness*. The main purpose of the intra-parcel correctness is to make sure that the datapath of the implementation meets the specification. The intra-parcel correctness also covers some aspects related to parcels flow. For instance, it ensures that a parcel is not lost, duplicated, or created, and guarantees that a parcel is steered to the right stage. The second criterion addresses the interaction between parcels and guarantees that inter-parcel dependencies are preserved. We refer to this aspect as the *inter-parcel (dependency) correctness*. The goal of the inter-parcel correctness is to ensure that both control and data flow in the implementation are consistent with the specification.

The intra-parcel correctness states that: if the final values read by an implementation parcel p_i are the same as those read by the corresponding specification parcel p_s, then the final values written by p_i shall be identical to those written by p_s. The intra-parcel correctness is formally expressed as follows:

$$\begin{aligned}
&\forall\, \sigma_i, p_i, p_s.\ \sigma_i \in \mathit{Runs}(\mathcal{I}).\ \exists\, \sigma_s \in \mathit{Runs}(\mathcal{S}).\\
&\quad p_i \ \overset{\sigma_i \sigma_s}{\underset{\mathbf{PCL}}{=\!=}}\ p_s\\
&\quad \wedge (\forall\, v_a, d.\ \mathrm{FRd}^{v_a}\ d\ p_i\ \sigma_i \iff \sigma_s^{p_s}.v_a = d)\\
&\quad \Longrightarrow \forall\, v'_a, d'.\ \mathrm{FWr}^{v'_a}\ d'\ p_i\ \sigma_i \iff \mathrm{SWr}^{v'_a}\ d'\ p_s\ \sigma_s
\end{aligned}$$

The inter-parcel correctness can be viewed as a protocol for parcel communications which guarantees that parcel dependencies are correctly handled by the implementation. This protocol takes the form of two rules that address the *direct-dependency* and the *no-dependency* situations explained in section 3.3. Both rules rely on the way parcels are ordered in the implementation and make no reference to the specification. Consequently, no reference model is needed for verifying these two rules.

We refer to the first rule as the *producer-consumer rule*. This rule addresses the case when there is a direct dependency from a parcel p_1 to a parcel p_2 with respect to an architectural variable v_a. The rule ensures that there is a physical variable v_i through which the data is passed from p_1 to p_2 properly. In other words, p_1 writes the final value of v_a to v_i, p_2 makes its final read of v_a from v_i, and no other parcel writes to v_i in between. The direct-dependency rule is formally stated as follows:

$$
\begin{aligned}
&\forall\, \sigma_i, v_a, p_1, p_2.\ \sigma_i \in \mathit{Runs}(\mathcal{I}).\\
&\quad \text{Retire } p_1\ \sigma_i \wedge \text{Retire } p_2\ \sigma_i \wedge p_1 \prec p_2\\
&\quad \wedge \text{Wr}^{v_a}\ p_1\ \sigma_i \wedge \text{Rd}^{v_a}\ p_2\ \sigma_i \wedge \forall\, p.\ p_1 \prec p \prec p_2.\ \neg(\text{Wr}^{v_a}\ p\ \sigma_i)\\
&\quad \Longrightarrow\\
&\quad \exists\, v_i, x, y.\\
&\qquad \text{FWr}^{v_a}_{v_i}\ p_1\ \sigma_i\ x \wedge \text{FRd}^{v_a}_{v_i}\ p_2\ \sigma_i\ y \wedge \forall\, z.\ x < z < y.\ \neg(\text{Wr}_{v_i}\ \sigma_i^z)
\end{aligned}
$$

The second rule is called the *no-producer rule*. This rule considers the case in which a parcel p does not depend on any older parcel with respect to an architectural variable v_a. The rule guarantees that p shall make its final read of v_a from a physical variable v_i to which no parcels make any writes. The no-dependency rule is expressed as follows:

$$
\begin{aligned}
&\forall\, \sigma_i, v_a, p.\ \sigma_i \in \mathit{Runs}(\mathcal{I}).\\
&\quad \text{Retire } p\ \sigma_i\\
&\quad \wedge \text{Rd}^{v_a}\ p\ \sigma_i \wedge \forall\, p' \prec p.\ \neg(\text{Wr}^{v_a}\ p'\ \sigma_i)\\
&\quad \Longrightarrow\\
&\quad \exists\, v_i, x.\\
&\qquad \text{FRd}^{v_a}_{v_i}\ p\ \sigma_i\ x \wedge \forall\, y < x.\ \neg(\text{Wr}_{v_i}\ \sigma_i^y)
\end{aligned}
$$

Specifying and verifying the inter-parcel correctness is one of the main contributions of this thesis. In section 3.6, we provide another version of the inter-parcel correctness that can be used in verifying implementations with abstract datapaths. Through the case study in chapter 4, we illustrate how the inter-parcel correctness can be verified in the context of microprocessors.

3.6 Specifying Inter-Parcel Correctness

This section introduces the criteria we use in determining whether a pipelined implementation preserves the dependencies between parcels. The criteria are formulated as two properties that correspond to the two inter-parcel rules presented in section 3.5. The first property describes the interaction between any two parcels where the leading parcel produces data to be consumed by the trailing parcel. In the second property, we address the case in which a parcel consumes data that is not produced by any leading parcel. The first is called *producer-consumer* property while the second is called the *no-producer* property.

The purpose of the inter-parcel rules is to define correctness from a mathematical perspective. The rules emphasize clarity and generality, and make use of infinite streams, comparison of data values across distant points in time and other features that are difficult to verify automatically. To simplify verification, we introduce additional instrumentation that allows us to translate these rules about infinite streams into properties about individual states that do not refer to data values.

We introduce instrumentation predicates to represent complex expressions in rules. The first category of complex expressions makes reference to data values. By replacing references to data values with instrumentation predicates, we reduce verification complexity by enabling the datapath to be abstracted away. The second category of complex expressions describe complicated sequences of events. Using an instrumentation predicate, verifying a rule is decomposed into two simpler tasks: (1) verifying the rule with the predicate in place of the complex expression and (2) verifying that the behavior of the predicate is consistent with the expression that it replaces.

We introduce three instrumentation predicates. The definitions of these predicates will vary from pipeline to pipeline. With each predicate, we give a criterion that the predicate must satisfy to ensure that the definition of the predicate is consistent with the intention. For an instrumented pipeline $\mathcal{I}$ as described in section 3.5, the predicates are:

1. Direct-dependency predicate ($\text{DDep} \subseteq V_a \times P \times P \times Q$): marks a direct-dependency between two parcels with respect to an architectural variable. For instance, $\text{DDep}^{v_a}\ p_1\ p_2\ q$ means that, at a state q, there is a direct-dependency from a parcel p_1 to a parcel p_2 with respect to an architectural variable v_a. In other words, v_a is both a destination of p_1 and a source of p_2, and v_a is not a destination of any parcel that comes in between p_1 and p_2, and eventually retires. The direct-dependency predicate should be defined such that:
$$\begin{aligned} &\forall\, \sigma_i, v_a, p_1, p_2, x.\ \sigma_i \in \mathit{Runs}(\mathcal{I}). \\ &\qquad \text{Retire}\ p_2\ \sigma_i^x \wedge \text{DDep}^{v_a}\ p_1\ p_2\ \sigma_i^x \\ &\qquad \Longleftrightarrow \\ &\qquad \text{Retire}\ p_1\ \sigma_i \wedge \text{Retire}\ p_2\ \sigma_i \wedge p_1 \prec p_2 \\ &\qquad \wedge\ \text{Wr}^{v_a}\ p_1\ \sigma_i \wedge \text{Rd}^{v_a}\ p_2\ \sigma_i \\ &\qquad \wedge\ \forall\, p.\ p_1 \prec p \prec p_2.\ \text{Retire}\ p\ \sigma_i \Longrightarrow \neg(\text{Wr}^{v_a}\ p\ \sigma_i) \end{aligned}$$

2. No-dependency predicate ($\text{NoDep} \subseteq V_a \times P \times Q$): holds when a parcel does not depend on any leading parcels with respect to an architectural variable. For instance, $\text{NoDep}^{v_a}\ p\ q$ means that, at a state q, there is no dependency from any older parcel to a parcel p with respect to an architectural variable v_a. Meaning that v_a is a source of p_2 and not a destination of any parcel that comes before p_1, and eventually retires. The no-dependency predicate should be defined such that:

$$\forall\, \sigma_i, v_a, p, x.\ \sigma_i \in \mathit{Runs}(\mathcal{I}).$$
$$\text{Retire}\ p\ \sigma_i^x \wedge \text{NoDep}^{v_a}\ p\ \sigma_i^x$$
$$\Longleftrightarrow$$
$$\text{Retire}\ p\ \sigma_i$$
$$\wedge\ \text{Rd}^{v_a}\ p\ \sigma_i$$
$$\wedge\ \forall\, p' \prec p.\ \text{Retire}\ p\ \sigma_i \Longrightarrow \neg(\text{Wr}^{v_a}\ p'\ \sigma_i)$$

3. Mispredict predicate ($\text{Mp} \subseteq V_a \times P \times Q$): holds when a parcel signals mispredict on the value of an architectural variable. The mispredict predicate should be defined such that:

$$\forall\, \sigma_i, v_a, d, d', x, x'.\ \sigma_i \in \mathit{Runs}(\mathcal{I}).$$
$$\text{Wr}^{v_a}\ d\ p\ \sigma_i^x \wedge \text{Wr}^{v_a}\ d'\ p\ \sigma_i^{x'} \wedge d \neq d' \wedge x < x'$$
$$\Longrightarrow$$
$$\exists\, y.\ x < y \leq x'.\ \text{Mp}^{v_a}\ p\ \sigma_i^y$$

Figures 3.5 and 3.6 use two timing diagrams to describe the scenarios specified in the producer-consumer and no-producer properties respectively. The x-axis represents the states. The y-axis represents the predicates. The predicates are grouped based on the parcels and/or the variables that they share. The values of the predicates are denoted by circles. A solid circle denotes a value of true while a hollow circle denotes a value of false. For the phase predicates, we use the first letter to imply a true value, *i.e.*, a circle with the letter "I" implies that the predicate "Infl" is true, *etc*. At any given state, the value of a predicate can be either a precondition or a postcondition in the specified scenario. The value is a postcondition by default. All preconditions are marked.

In figure 3.5, we sketch the scenario specified in the producer-consumer property. The scenario marks some events of interest taking place during the lifetime of two arbitrary

parcels p_1 and p_2 representing the producer and the consumer respectively. The scenario highlights four preconditions that need to be satisfied in order to enforce the producer-consumer relationship between p_1 and p_2.

The scenario begins at some state q_{F_1} where p_1 enters the pipeline (first precondition) and shows that p_1 eventually becomes retired (second precondition). The scenario ends when the consumer parcel p_2 exits the pipeline at some state q_{R_2} (third precondition) where there is a direct dependency from p_1 to p_2 with respect to some architectural variable v_a (fourth precondition).

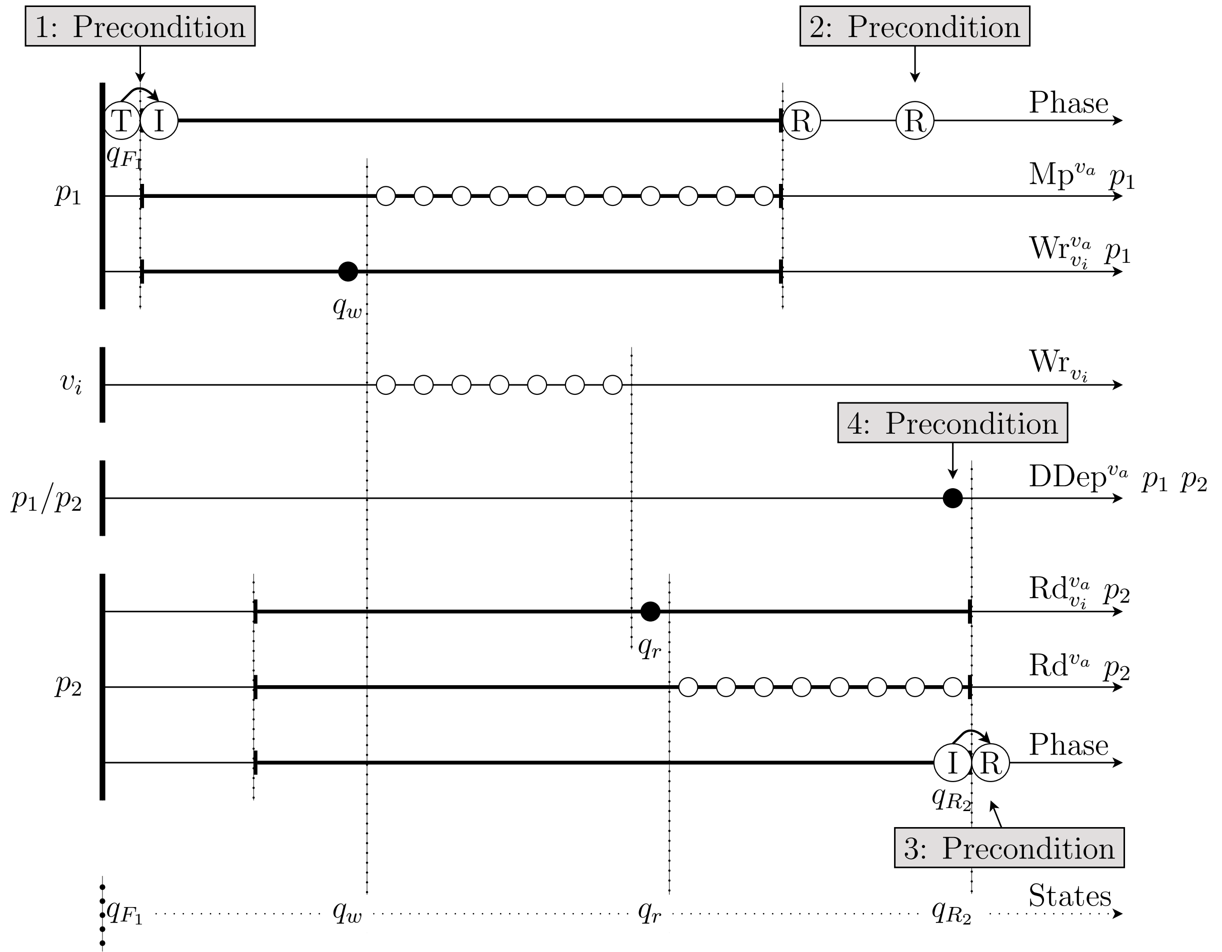

Figure 3.5: Producer-consumer property

As shown in figure 3.5, the key to satisfying the producer-consumer property is the existence of some physical variable v_i (mapped to v_a) through which the data is passed from the producer p_1 to the consumer p_2. The way the producer-consumer relationship is enforced in the property is threefold. First, the producer p_1 makes a final write to v_i at some state q_w. Second, the consumer p_2 makes a final read from v_i at some state q_r. Third, to insure the data is passed correctly from the producer to the consumer, no other parcel is allowed to write to v_i in between states q_w and q_r.

The producer-consumer property makes no restrictions on the way the producing and consuming parcels interact with the pipeline state prior to making their final write and read respectively. In other words, the producer p_1 may make an arbitrary number of speculative writes to those physical variables that are mapped to v_a before reaching q_w. Similarly, the consumer p_2 may speculatively read from the physical variables that are mapped to v_a for an arbitrary number of times before its final read at state q_r.

The producer-consumer property, sketched in figure 3.5, can be formally expressed in the linear temporal logic as follows:

$$
\begin{array}{l}
\textbf{PropProdCons} \equiv \\
\quad \forall\, v_a, p_1, p_2.\ \exists\, v_i. \\
\quad \mathbf{G} \left(\begin{array}{l}
\text{Fetch } p_1 \wedge \mathbf{X}\ \mathbf{F}\ \text{Ret } p_1 \\
\Longrightarrow \\
\mathbf{X} \left(\begin{array}{l}
\neg(\text{Retire } p_2 \wedge \text{DDep}^{v_a}\ p_1\ p_2) \\
\mathbf{W} \\
\left(\begin{array}{l}
\quad \text{Wr}^{v_a}_{v_i}\ p_1 \\
\wedge\, \mathbf{X}\ (\neg \text{Mp}^{v_a}\ p_1\ \mathbf{W}\ \text{Ret } p_1) \\
\wedge\, \mathbf{X} \left(\begin{array}{l}
\neg \text{Wr}_{v_i} \\
\mathbf{W} \\
\left(\begin{array}{l}
\quad \text{Rd}^{v_a}_{v_i}\ p_2 \\
\wedge\, \mathbf{X} \left(\begin{array}{l}
\neg \text{Rd}^{v_a}\ p_2 \\
\mathbf{W} \\
\text{Retire } p_2 \wedge \text{DDep}^{v_a}\ p_1\ p_2
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array}
$$

The no-producer property is pictorially represented in figure 3.6. The scenario sketched in the figure begins from the initial state q_0 and involves some events related to an arbitrary

consuming parcel p. The scenario has three preconditions. First, parcel p enters the pipeline at some state q_F. Second, p exits at some following state q_R. Third, at state q_R, p does not have any dependencies with respect to specification register v_a (third precondition).

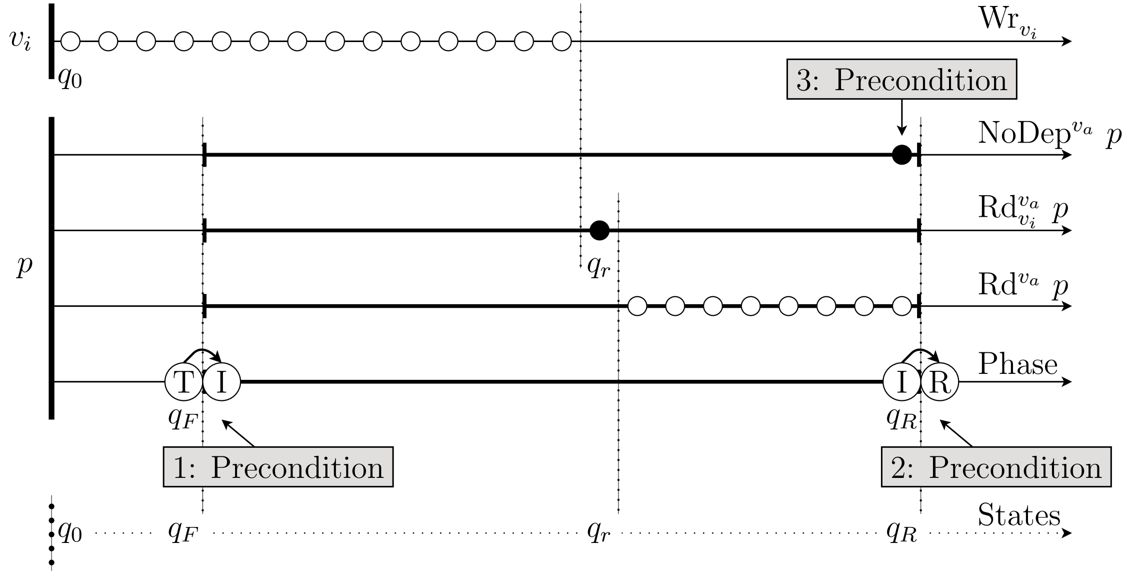

Figure 3.6: No-producer property

If the three preconditions are satisfied, the no-producer property guarantees the existence of some physical variable v_i from which p makes its final read for the value of v_a at some state q_r. It also guarantees that no parcels have written to v_i at any state before q_r. Hence, the initial value of v_i is kept unchanged until it gets consumed by p.

Similar to the producer-consumer property, the no-producer property allows the consumer p to freely interact with pipeline state before making its final read at q_r. The no-producer property is formally specified in the linear temporal logic as follows:

$$
\begin{array}{l}
\textbf{PropNoProd} \equiv \\
\quad \forall\, v_a, p.\ \exists\, v_i. \\
\qquad \left(\begin{array}{l}
\mathbf{F}\ (\mathrm{Fetch}\ p \wedge \mathbf{X}\ \mathbf{F}\ (\mathrm{Retire}\ p \wedge \mathrm{NoDep}^{v_a}\ p)) \\
\Longrightarrow \\
\left(\begin{array}{l}
\neg \mathrm{Wr}_{v_i} \\
\mathbf{W} \\
\left(\begin{array}{ll}
 & \mathrm{Rd}^{v_a}_{v_i}\ p \\
\wedge\ \mathbf{X} & \left(\begin{array}{l}
\neg \mathrm{Rd}^{v_a}\ p \\
\mathbf{W} \\
\mathrm{Retire}\ p \wedge \mathrm{NoDep}^{v_a}\ p
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}
$$

3.7 Decomposing Inter-Parcel Correctness

In section 3.6, the inter-parcel dependency correctness is presented in the form of two properties: **PropProdCons** and **PropNoProd**. Both of these properties address some key events in the lifetime of any parcel and describe the interaction between parcels that may or may not overlap in time. Due to the temporal complexity of properties **PropProdCons** and **PropNoProd**, we break them down into a set of smaller properties that are more suitable for model checking. In breaking down **PropProdCons** and **PropNoProd**, we introduce an extra predicate:

Source predicate ($\mathrm{Src} \subseteq V_a \times P \times Q$): shows whether an architectural variable is the source of a parcel.

In this section, we focus on explaining the properties resulting from the decomposition. The soundness of the decomposition is proven in section 3.8. We classify the properties presented here into two categories: *obligations* (subsection 3.7.1) and *consistency conditions* (subsection 3.7.2). The purpose of the obligations is to uncover bugs in the implementation. The consistency conditions on the other hand capture inconsistencies in predicates definitions and prevent vacuous verification. Although this classification is based on conceptual rather than syntactical differences, the consistency conditions are generally simpler than the obligations.

The decomposition of properties **PropProdCons** and **PropNoProd** is illustrated in figures 3.7 and 3.8 respectively. First, properties **PropProdCons** and **PropNoProd** are decomposed into 4 obligations and 11 consistency conditions. Then, two out of the four obligations (**Ob1** and **Ob3**) are further broken down into two (smaller) obligations and three consistency conditions.

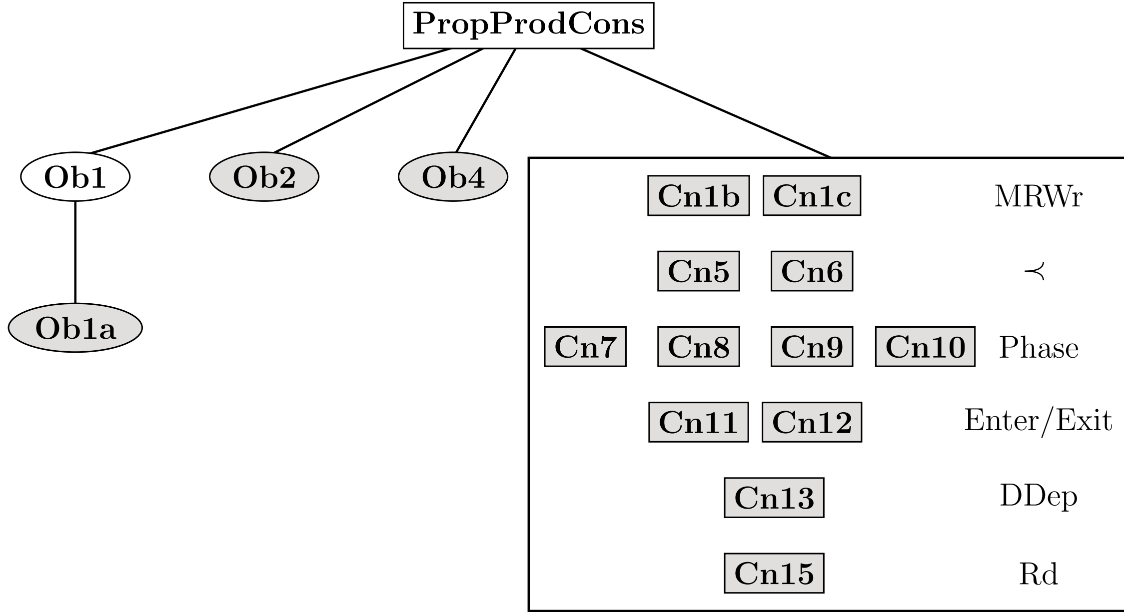

Figure 3.7: Decomposition tree of the producer-consumer property

3.7.1 Obligations

Obligation **Ob1** says that the producer does not signal mispredict between its final write and the consumer's final read. Obligation **Ob2** says that the producer does not signal mispredict after the consumer's final read. Together, these two obligations prevent bugs whereby a consumer reads an incorrect speculative value from a producer. In the case that a consumer is dependent upon the initial state (*i.e.*, is not dependent on any older parcel), obligation **Ob3** ensures that no parcel writes to the variable before the consumer does its final read.

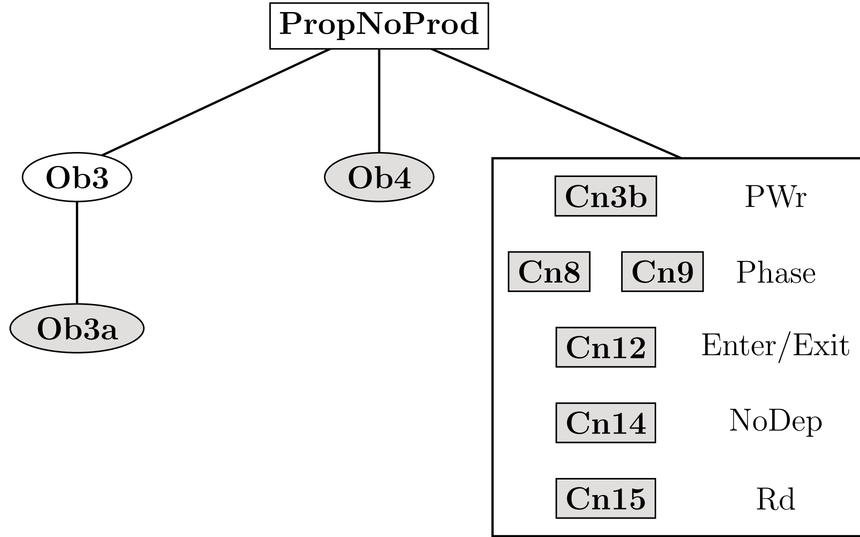

Figure 3.8: Decomposition tree of the no-producer property

Unlike the rest of the properties, obligations **Ob1** and **Ob3** address events in the past. To break down these obligation into smaller properties about present and future time, we introduce two additional predicates. The purpose of these predicates is to capture history information about the writes to physical variables. The two predicates can be described as follows:

1. Most-recent-write predicate: $\text{MRWr}_{v_i}^{v_a}\ p\ q$ means that the most recent write to a physical variable v_i has been made by a parcel p, and at the time of that write v_i was mapped to an architectural variable v_a.

2. Past-write predicate: $\text{PWr}_{v_i}^{v_a}\ p\ q$ means that a parcel p has written to a physical variable v_i at some point in the past and at the time of that write v_i was mapped to an architectural variable v_a.

Obligation **Ob1** is decomposed into obligation **Ob1a** and consistency conditions **Cn1b** and **Cn1c**. Obligation **Ob1a** uses the most-recent-write predicate to say that the consumer makes its final read from a variable that was written to most recently by the producer. The consistency conditions ensure that the most-recent-write predicate is consistent with the behavior of the pipeline.

Obligation **Ob3** is decomposed into obligation **Ob3a** and consistency condition **Cn3b**. Obligation **Ob3a** uses the past-write predicate to ensure that no parcel has written to the variable from which the consumer does its final read. The consistency condition describes the required characteristics of the past-write predicate.

The final obligation (obligation **Ob4**) says that if a parcel should read an architectural variable, then it performs the read before retirement. The obligations are described in more detail over the rest of this subsection.

Ob1[‡]: **No misprediction is signaled between producer's final write and consumer's final read.**

If a parcel p_2 makes its final read for an architectural variable v_a using a physical variable v_i, and at the time p_2 retires there exists a direct dependency from a parcel p_1 to p_2 with respect to v_a, then before the current state there exists a write for the value of v_a made to v_i by p_1. From that state at which p_1 writes to v_i, p_1 shall not signal mispredict on the value of v_a.

$$\forall\, v_i, v_a, p_1, p_2.$$
$$\mathbf{G}\left(\begin{array}{l} \mathrm{Rd}^{v_a}_{v_i}\ p_2 \\ \wedge\ \mathbf{X}\ (\neg\mathrm{Rd}^{v_a}\ p_2\ \mathbf{U}\ (\neg\mathrm{Rd}^{v_a}\ p_2 \wedge \mathrm{Retire}\ p_2 \wedge \mathrm{DDep}^{v_a}\ p_1\ p_2)) \\ \Longrightarrow \\ \hat{\mathbf{X}}\ ((\neg\mathrm{Wr}^{v_a} \wedge \neg\mathrm{Mp}^{v_a}\ p_1 \wedge (\mathrm{Infl}\ p_1 \vee \mathrm{Ret}\ p_1))\ \hat{\mathbf{U}}\ (\mathrm{Wr}^{v_a}_{v_i}\ p_1 \wedge \mathrm{Infl}\ p_1)) \end{array}\right)$$

Ob2: **No misprediction is signaled after consumer's final read.**

If a parcel p_2 makes a final read for the value of an architectural variable v_a using a physical variable v_i, and at the time p_2 retires there exists a direct dependency from a parcel p_1 to p_2 with respect to v_a, then p_1 must not signal mispredict on the value of v_a.

$$\forall\, v_i, v_a, p_1, p_2.$$
$$\mathbf{G}\left(\begin{array}{l} \mathrm{Rd}^{v_a}_{v_i}\ p_2 \\ \wedge\ \mathbf{X}\ (\neg\mathrm{Rd}^{v_a}\ p_2\ \mathbf{U}\ (\neg\mathrm{Rd}^{v_a}\ p_2 \wedge \mathrm{Retire}\ p_2 \wedge \mathrm{DDep}^{v_a}\ p_1\ p_2)) \\ \Longrightarrow \\ \neg\mathrm{Mp}^{v_a}\ p_1\ \mathbf{W}\ \mathrm{Ret}\ p_1 \end{array}\right)$$

[‡]**Ob1** is decomposed into **Ob1a**, **Cn1b** and **Cn1c**.

Ob3[§]: **No writes happen before consumer's final read.**

If a parcel p_2 makes a final read for the value of an architectural variable v_a using a physical variable v_i, and at the time p_2 retires it has no dependencies with respect to v_a, there exists no writes to v_i prior to the current state.

$\forall\, v_i, v_a, p.$

$$\mathbf{G}\left(\begin{array}{l}\mathrm{Rd}^{v_a}_{v_i}\ p \\ \wedge\ \mathbf{X}\ (\neg\mathrm{Rd}^{v_a}\ p\ \mathbf{U}\ (\neg\mathrm{Rd}^{v_a}\ p \wedge \mathrm{Retire}\ p \wedge \mathrm{NoDep}^{v_a}\ p)) \\ \Longrightarrow \\ \hat{\mathbf{X}}\ \hat{\mathbf{G}}\ \neg\mathrm{Wr}^{v_a}\end{array}\right)$$

Ob4: **Source is read before retirement.**

If a parcel p enters the pipeline, then starting from the next state, p shall not retire before reading v_a if v_a is marked as its source at retirement time.

$\forall\, p.$

$\mathbf{G}$ (Fetch $p \Longrightarrow \mathbf{X}$ $(\neg(\mathrm{Retire}\ p \wedge \mathrm{Src}^{v_a}\ p)\ \mathbf{W}\ \mathrm{Rd}^{v_a}\ p))$

Ob1a: **Consumer makes its final read from a variable where the most recent write to that variable is made by producer.**

If a parcel p_2 makes a final read for the value of an architectural variable v_a using a physical variable v_i, and at the time p_2 retires there exists a direct dependency from a parcel p_1 to p_2 with respect to v_a, then the most recent write to v_i must have been done by p_1.

$\forall\, v_i, v_a, p_1, p_2.$

$$\mathbf{G}\left(\begin{array}{l}\mathrm{Rd}^{v_a}_{v_i}\ p_2 \\ \wedge\ \mathbf{X}\ (\neg\mathrm{Rd}^{v_a}\ p_2\ \mathbf{U}\ (\neg\mathrm{Rd}^{v_a}\ p_2 \wedge \mathrm{Retire}\ p_2 \wedge \mathrm{DDep}^{v_a}\ p_1\ p_2)) \\ \Longrightarrow \\ \mathrm{MRWr}^{v_a}_{v_i}\ p_1\end{array}\right)$$

Ob3a: **Consumer's final read is made from a variable to which no parcel has written.**

If a parcel p_2 makes a final read for the value of an architectural variable v_a using a

[§] **Ob3** is decomposed into **Ob3a** and **Cn3b**.

physical variable v_i, and at the time p_2 retires it has no dependencies with respect to v_a, then in the current state v_i shall not be marked with any past writes.
$\forall\, v_i, v_a, p.$

$$\mathbf{G}\left(\begin{array}{l}\mathrm{Rd}^{v_a}_{v_i}\ p \\ \wedge\ \mathbf{X}\ (\neg\mathrm{Rd}^{v_a}\ p\ \mathbf{U}\ (\neg\mathrm{Rd}^{v_a}\ p \wedge \mathrm{Retire}\ p \wedge \mathrm{NoDep}^{v_a}\ p)) \\ \Longrightarrow \\ \neg\mathrm{PWr}_{v_i}\end{array}\right)$$

3.7.2 Consistency Conditions

Consistency conditions **Cn1b**, **Cn1c** and **Cn3b** describe the relationship between the write predicate, and the most-recent-write and past-write predicates. Together, conditions **Cn1b** and **Cn1c** guarantee that the most-recent-write predicate is reset when the parcel is fetched, becomes true after the parcel writes, and stays true unless another parcel writes or a misprediction is signaled. Condition **Cn3b** ensures that the past-write predicate holds once the parcel writes.

The main characteristics of the parcel order are described by consistency conditions **Cn5** and **Cn6**. The first condition states that parcel order shall be fixed over time. The second condition says that when a parcel is fetched, no younger parcels shall be in-flight.

Consistency conditions **Cn7** through **Cn10** address the relationship between different phase predicates. Condition **Cn7** says that an in-flight parcel cannot be in the retired phase. Conditions **Cn8** and **Cn9** imply that retired parcels and discarded parcels never become in-flight. Similarly, condition **Cn10** says that a discarded parcel cannot become retired.

Consistency conditions **Cn11** and **Cn12** describe how the phase of a parcel changes when it enters or exits the pipeline respectively. Condition **Cn11** says that a parcel needs to be in the top phase before it becomes in-flight. Condition **Cn12** states that when a parcel ceases to be in-flight, it either becomes retired or discarded.

The dependency predicates are the focus of consistency conditions **Cn13** and **Cn14**. Condition **Cn13** says that when there is a direct dependency between two parcels on one of the architectural variables, the producer shall be the older parcel and the variable shall

be the source of the consumer. Condition **Cn14** states that if the no-dependency predicate holds for a parcel on one of the architectural variables, the variable shall be the source of that parcel.

The last consistency condition **Cn15** ensures that when an architectural variable is read by a parcel, there exists a corresponding physical variable from which the read is made. The rest of this subsection has more details about the consistency conditions.

Cn1b: **Most-recent-write predicate is set to false at fetch time.**

If at the current state a parcel p enters the pipeline, then in the next state p shall not be marked as the parcel which made the most recent write to a physical variable v_i which is mapped to an architectural variable v_a.
$\forall\, v_i, v_a, p.$

$\mathbf{G}\ (\text{Fetch } p \Longrightarrow \mathbf{X}\ \neg\text{MRWr}_{v_i}^{v_a}\ p)$

Cn1c: **Most-recent-write predicate becomes true after write and stays true unless another parcel writes or a misprediction is signaled.**

If in the next state a parcel p is marked as the parcel making the most recent writer to a physical variable v_i mapped to an architectural variable v_a, then in the current state, either p writes to v_i, or else three conditions shall hold: p is marked as the parcel making the most recent write to v_i, p does not signal mispredict on the value v_a, and no other parcel writes to v_i.
$\forall\, v_i, v_a, p.$

$\mathbf{G}\ (\mathbf{X}\ \text{MRWr}_{v_i}^{v_a}\ p \Longrightarrow \text{Wr}_{v_i}^{v_a}\ p \vee (\text{MRWr}_{v_i}^{v_a}\ p \wedge \neg\text{Mp}^{v_a}\ p \wedge \neg\text{Wr}_{v_i}))$

Cn3b: **Past-write predicate is set to true after any write.**

If a parcel writes to a physical variable v_i or v_i is marked with a past write, then v_i shall be marked with a past write in the next state.
$\forall\, v_i.$

$\mathbf{G}\ ((\text{PWr}_{v_i} \vee \text{Wr}_{v_i}) \Longrightarrow \mathbf{X}\ \text{PWr}_{v_i}\ )$

Cn5: **Parcel order does not change over time.**

If in the next state a parcel p_1 comes in order before a parcel p_2, the same shall be true in the current state.
$\forall\, p_1, p_2.$

$\mathbf{G}\ (\mathbf{X}\ p_1 \prec p_2 \Longrightarrow p_1 \prec p_2)$

Cn6: **None of the younger parcels are in-flight at fetch time.**

If a parcel p_1 enters the pipeline, in the next state the phase of every younger parcel p_2 shall not be in-flight.
$\forall\, p_1, p_2.$
$\mathbf{G}$ (Fetch $p_1 \wedge p_1 \prec p_2 \Longrightarrow \mathbf{X}\ \neg$Infl p_2)

Cn7: **In-flight implies not retired.**

If a parcel p is in the in-flight phase, it can not be in the retired phase.
$\forall\, p.$
$\mathbf{G}$ (Infl $p \Longrightarrow \neg$Ret p)

Cn8: **Retired parcel never becomes in-flight.**

If a parcel p is in the retired phase, it never becomes in-flight.
$\forall\, p.$
$\mathbf{G}$ (Ret $p \Longrightarrow \mathbf{G}\ \neg$Infl p)

Cn9: **Discarded parcels never become in-flight.**

If a parcel p is in the discarded phase, it never becomes in-flight.
$\forall\, p.$
$\mathbf{G}$ (Dis $p \Longrightarrow \mathbf{G}\ \neg$Infl p)

Cn10: **Discarded parcels never become retired.**

If a parcel p is in the discarded phase, it shall not be in the retired phase.
$\forall\, p.$
$\mathbf{G}$ (Dis $p \Longrightarrow \mathbf{G}\ \neg$Ret p)

Cn11: **Only parcels in top may become in-flight.**

For a parcel p to change its phase to become in-flight in the next state, it has to be currently in the top phase.
$\forall\, p.$
$\mathbf{G}$ ($\neg$Infl $p \wedge \mathbf{X}$ Infl $p \Longrightarrow$ Top p)

Cn12: **In-flight changes to retired or discarded.**

If an in-flight parcel p changes its phase in the next state, it becomes either retired or discarded.
$\forall\, p.$
$\mathbf{G}\ (\text{Infl}\ p \wedge \mathbf{X}\ \neg\text{Infl}\ p \Longrightarrow \mathbf{X}\ (\text{Ret}\ p \vee \text{Dis}\ p))$

Cn13: **Direct-dependency predicate is stronger than source predicate.**

If there is a direct dependency from one parcel p_1 to another parcel p_2 with respect to an architectural variable v_a at the time p_2 retires, then p_1 shall come before p_2 in order and v_a shall be a (final) source of p_2.
$\forall\, v_a, p_1, p_2.$
$\mathbf{G}\ (\text{DDep}^{v_a}\ p_1\ p_2 \wedge \text{Retire}\ p_2 \Longrightarrow p_1 \prec p_2 \wedge \text{Src}^{v_a}\ p_2)$

Cn14: **No-dependency predicate is stronger than source predicate.**

If a parcel p has no dependencies with respect to an architectural variable v_a at the time it retires, then v_a shall be marked as a (final) source of p.
$\forall\, v_a, p.$
$\mathbf{G}\ (\text{NoDep}^{v_a}\ p \wedge \text{Retire}\ p \Longrightarrow \text{Src}^{v_a}\ p)$

Cn15: **Reading an architectural variable is made from a physical variable mapped to it.**

If a parcel p reads an architectural variable v_a, there exists a physical variable v_i from which p reads the value of v_a and v_i is mapped to v_a.
$\forall\, v_a, p.$
$\mathbf{G}\ (\text{Rd}^{v_a}\ p \Longrightarrow \exists\, v_i.\ \text{Rd}^{v_a}_{v_i}\ p)$

3.8 Soundness of Decomposition

In this section, we show that verifying the obligations and consistency conditions presented in section 3.7 guarantees the satisfaction of the inter-parcel dependency properties defined in section 3.6, namely, the producer-consumer (**PropProdCons**) and the no-producer

(**PropNoProd**) properties. More precisely, we justify the soundness of the decomposition trees associated with the two inter-parcel dependency properties and shown in figures 3.7 and 3.8 respectively.

In figure 3.9, we sketch a proof to show that the obligations imply the producer-consumer property (**PropProdCons**). The premises of our proof are the four preconditions of **PropProdCons** (steps 1-4). Given the obligations and the premises, we prove that there exists a physical variable v_i (representing the architectural variable v_a) through which the producing parcel p_1 passes uncorrupted data to the consuming parcel p_2.

We first show that v_a shall be marked as a final source of p_2 at the time it retires (step 5). At that time, p_1 must come in order before p_2 (step 6) and that is true also at the time p_1 enters the pipeline (step 7). Consequently, at the time p_1 becomes in-flight, p_2 shall not be in-flight (step 8). p_2 enters the pipeline in a following state (step 9) and its phase turns in-flight and stays so until it retires (step 10). Before it retires, p_2 has to make a final read of the value of v_a (step 11) through some physical variable v_i (step 12).

Similarly, once p_1 enters the pipeline its phase becomes in-flight and stays so until it exits (step 13). During that window and before p_2 makes its final read, p_1 shall make a final write for the value of v_a to v_i (step 14). Between the write and the read no parcel writes to v_i (step 15). After the write, p_1 does not signal mispredict on the value of v_a (steps 16-17).

Our proof for the soundness of decomposing the no-producer property (**PropNoProd**) is sketched in figure 3.10. The proof premises are the three preconditions of **PropNoProd**. The goal is to show that satisfying the preconditions, the obligations, and the consistency conditions guarantees the existence of a physical variable v_i (representing the architectural variable v_a) whose initial value is not altered by any parcel and from which the consuming parcel p_2 makes its final read.

First, we show that the phase of p_2 has to be in-flight as long as p_2 is being processed by the pipeline (step 4). At the time p_2 retires, v_a shall be its final source (step 5). Therefore, before its retirement, p_2 makes a final read for the value of v_a (step 6) from one of the physical variables, namely v_i (step 7). Finally, it is shown that before the read takes place, no parcel writes to v_i (step 8).

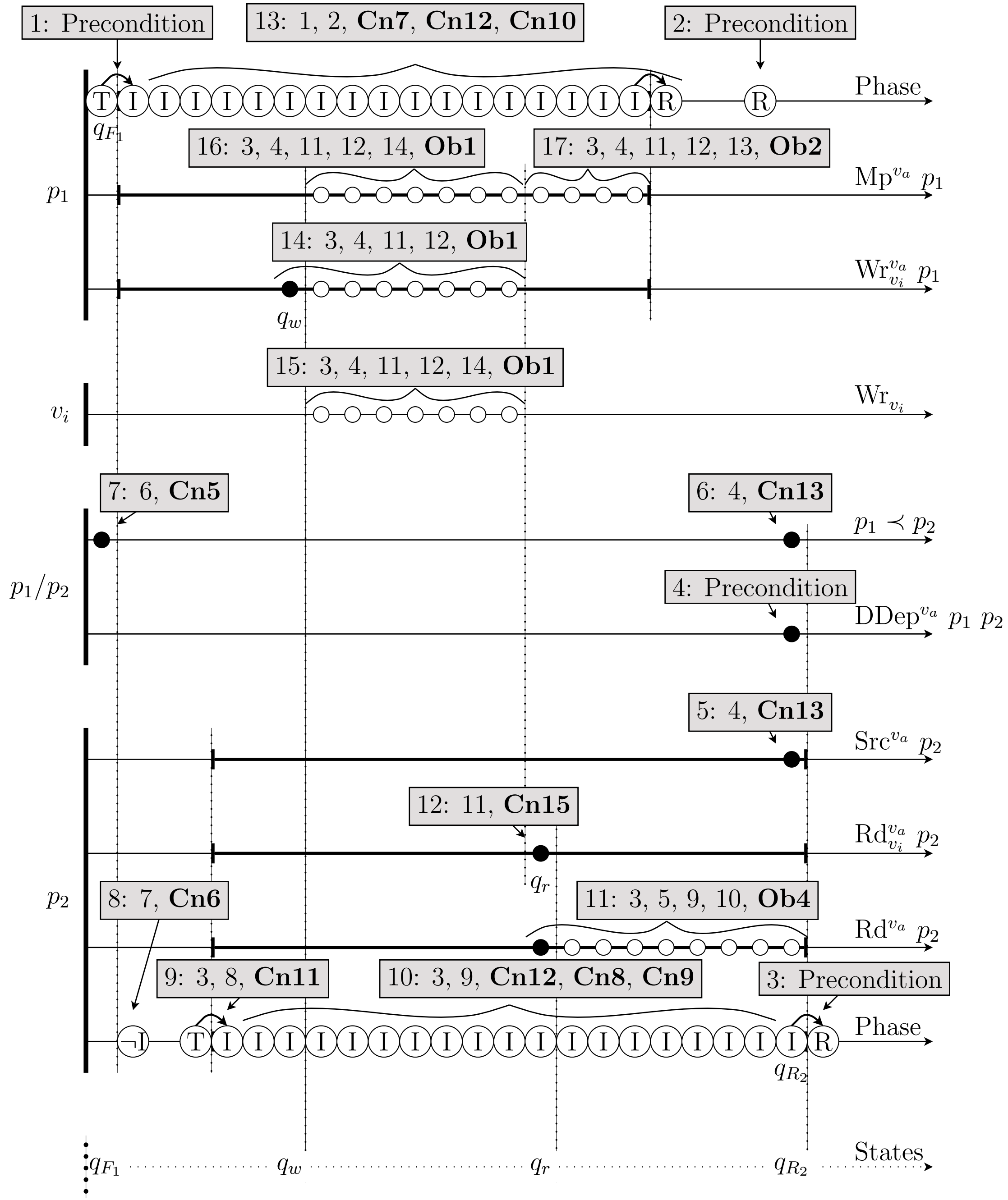

Figure 3.9: Sketch of decomposition proof of the producer-consumer property

Figure 3.10: Sketch of decomposition proof of the no-producer property

3.9 Summary

Correctness of a pipeline is specified in terms of the behavior of its parcels. A pipeline is instrumented with predicates to monitor some parcel activities such as reading from or writing to the physical variables. Using this instrumentation, inter-parcel correctness is expressed in the form of two properties that support speculative out-of-order processing of parcels. The two properties are decomposed into 4 obligations and 14 consistency conditions. The obligations detect implementation bugs while the consistency conditions ensure instrumentation predicates are defined correctly. The decomposition is proven to be sound.

Chapter 4

Processor Case Study

We conducted a case study to illustrate our verification techniques (presented in chapters 2 and 3) and evaluate their effectiveness. We implemented a tool, named Tahrir, aimed at verifying syntactically-safe LTL (SSLTL) properties using the algorithm explained in section 2.2. We used Tahrir to verify that the inter-parcel properties introduced in section 3.6 are satisfied by a processor. The processor chosen for our case study supports speculative out-of-order execution of instructions. Structural hazards and functionality of the execution units are abstracted away from the processor model since the focus is on verifying inter-instruction dependencies.

Tahrir is introduced in section 4.1. The microarchitecture of the processor is presented in section 4.2. The processor model is explained in section 4.3. The verification is described in section 4.4. An analysis of the verification can be found in section 4.5. The chapter is summarized in section 4.6.

4.1 SSLTL Verification Tool - Tahrir

In this section, we present *Tahrir**, a tool aimed at inductively verifying syntactically-safe LTL (SSLTL) properties about infinite-state systems. Tahrir implements the algorithm presented in section 2.2 (function *Verify*). Tahrir has a generic interface at its core that allows using an SMT solver (or invariant checker) as a decision procedure. Both CVC3 [3] and UCLID [6] are currently supported. Apart from its decision engine, Tahrir is implemented in more than 6000 lines of Moscow ML [50] code.

Tahrir takes as input a model to be verified and a set of SSLTL properties about that model. In addition to that, the user needs to set the induction depth (k) and choose which decision procedure is to be used by Tahrir. For each property p, the user needs to provide a proof statement through which the arguments of p (if they exist) can be bound with a universal quantifier. The proof statement includes lists of *lemmas* and *assumptions* to be used in proving the target invariant (*i.e.*, e_p in section 2.2) by induction. In the base case of the induction, the assumptions are used to prove that the model satisfies the target invariant in the initial $k-1$ states. In the inductive case, the assumptions and lemmas are combined to show that the model satisfy the target invariant in the kth step.

The following is an example of a proof statement:

```
PSTMT1:  PROVE FORALL(x,y). p(x,y)
         USING FORALL(v,w). l1(v),l2(x,y),l3(v,w)
         ASSUMING FORALL(u). a1(u),a2(x)
```

where `p` is the (target) property to be verified using $\mathtt{l}_1$, $\mathtt{l}_2$ and $\mathtt{l}_3$ as lemmas, and $\mathtt{a}_1$ and $\mathtt{a}_2$ as assumptions. Typically, both the assumptions and lemmas are SSLTL formulas of the form $\mathbf{G}\ \phi$ where ϕ is purely combinational (*i.e.*, ϕ does not contain any temporal operators). However, the use of generic SSLTL formulas instead is not syntactically restricted.

Generally, a proof statement may contain up to three sets of variables bound by universal quantification. In the case of `PSTMT1`, these three sets are $\{\mathtt{x},\mathtt{y}\}$, $\{\mathtt{v},\mathtt{w}\}$, and $\{\mathtt{u}\}$. We define the *number of outer quantifiers* (NOQ) to be the number of variables in the first set

*Our tool is named after *Tahrir Square* (Liberation Square), a major public town square in downtown Cairo, Egypt. Tahrir Square was the epicenter of the Egyptian revolution triggered on January 25, 2011.

while the *number of inner quantifiers* (NIQ) to be the number of variables that belong to the other two sets. For instance, The numbers NOQ and NIQ for proof statement `PSTMT1` are two and three respectively.

The modeling language of Tahrir is based on that of UCLID. The language has four native datatypes: `TRUTH` (Boolean), `TERM` (unbounded integers), `PRED[n]` (n-ary predicates where $\mathtt{n} \in \mathbb{N}^+$), and `FUNC[n]` ($n$-ary functions where $\mathtt{n} \in \mathbb{N}^+$). Creating enumerated datatypes is also supported in the language. Both native and enumerated datatypes can be used in defining the model variables and inputs with the exception that inputs have to be of 0-ary datatypes. Symbolic constants can be defined using native datatypes.

Expressions of type `TRUTH` can be built using relational operators (*e.g.*, $<$ and $>$), Boolean operators (*e.g.*, $\wedge$ and $\vee$), and predicate applications (*e.g.*, $\mathtt{p}(\mathtt{x}, \mathtt{y})$ where `p` is of type `PRED[2]`). Function applications can be used in constructing expressions of type `TERM` as well as enumerated types. Expressions for predicates and functions are represented as lambda expressions (*e.g.*, $\mathtt{Lambda}(\mathtt{x}, \mathtt{y}).\mathtt{x} > \mathtt{y}$).

The behavior of the model is specified as a set of expressions assigned to its variables. Each combinational variable is assigned a single expression that represents its value in the current state. Each state variable can be assigned two expressions: one to represent its initial value and another to represent its value in the next state.

The ultimate goal of Tahrir is to check whether the language of the model is contained in the language of the Büchi automaton that is constructed from an SSLTL property. To achieve this goal, Tahrir follows the algorithm explained in section 2.2. First, Tahrir translates the property into a Büchi automaton. Then, for each state in the Büchi automaton, the model is augmented with a Boolean (history) variable to keep track of that state. This encoding of states (as Boolean variables) is used because the automaton is non-deterministic and the encoding allows us easily to represent the automaton being in multiple states at the same time.

Based on Theorem 2.1, for language containment to hold, it shall be always the case that at least one of the history variables, representing the states of the generated Büchi automaton, is true. We refer to this as the *target invariant* (*i.e.*, e_p in section 2.2). For SSLTL properties of the form $\mathbf{G}\ e$, where e does not contain any temporal operators, Tahrir

considers e to be the target invariant. For this type of property, Tahrir does not generate a Büchi automaton and adds no history variables to the model.

With other forms of properties (*i.e.*, other than $\mathbf{G}\ e$), Tahrir constructs the Büchi automaton using the basic algorithm proposed by Gerth *et al.* [18]. After that Tahrir tries to minimize the size of the automaton by merging the states which have identical predecessors or successors. During this step, Tahrir repeatedly calls the SAT solver zChaff [46] to identify pairs of matching states. The minimized version of the automaton is then used to generate the target invariant.

Tahrir verifies whether the target invariant is indeed an invariant using k-step induction. In the base case, Tahrir simulates the model for $k-1$ steps from the initial states and calls the SMT solver to check whether the target invariant is satisfied in each step. The actual Boolean formula that gets checked by the SMT solver is generated taking into consideration the list of assumptions (specified in the proof statement) as well as the target invariant. For instance, the Boolean formula that gets generated in the base case for proof statement `PSTMT1` is:

$$\forall\, \mathtt{x}, \mathtt{y}.\ (\forall\, \mathtt{u}.\ \mathtt{a}_1|_0^{\mathtt{k}-1}(\mathtt{u}) \wedge \mathtt{a}_2|_0^{\mathtt{k}-1}(\mathtt{x})) \Longrightarrow \mathtt{p}|_0^{\mathtt{k}-1}(\mathtt{x}, \mathtt{y})$$

where $\phi|_m^n$ represents the invariant generated from a property ϕ and expanded over steps m through n. In other words, it is the conjunction of all the instances of the invariant from step m to step n.

In the inductive case, Tahrir simulates the model for k steps starting from an unconstrained state. The SMT solver is called to check whether the target invariant is satisfied in the kth step. The Boolean formula passed to the SMT solver includes the lemmas as induction hypotheses to limit induction within the reachable state space. For instance, the Boolean formula that gets generated in the inductive case for proof statement `PSTMT1` is:

$$\begin{aligned}\forall\, \mathtt{x}, \mathtt{y}.\quad & (\forall\, \mathtt{u}.\ \mathtt{a}_1|_0^{\mathtt{k}}(\mathtt{u}) \wedge \mathtt{a}_2|_0^{\mathtt{k}}(\mathtt{x})) \\ & \wedge (\forall\, \mathtt{v}, \mathtt{w}.\ \mathtt{l}_1|_0^{\mathtt{k}-1}(\mathtt{v}) \wedge \mathtt{l}_2|_0^{\mathtt{k}-1}(\mathtt{x}, \mathtt{y}) \wedge \mathtt{l}_3|_0^{\mathtt{k}-1}(\mathtt{v}, \mathtt{w})) \\ & \Longrightarrow \mathtt{p}|_{\mathtt{k}}^{\mathtt{k}}(\mathtt{x}, \mathtt{y})\end{aligned}$$

For flexibility, the target invariant is not considered as an induction hypothesis by default, but instead, the user may include the target property (*e.g.*, `p` in the case of `PSTMT1`) as one of the lemmas in the proof statement.

4.2 Processor Microarchitecture

Our aim in this section is to describe the microarchitecture of the processor to which we apply our verification strategies presented in chapters 3 and 2. The microarchitecture used in our case study supports two features found in most modern microprocessors: *out-of-order* and *speculative* execution of instructions. We focus here on these two features because of their strong impact on the way instruction dependencies are handled within a microprocessor.

The microarchitecture presented here can be viewed as a six-stage pipeline. Some of these pipeline stages may be pipelined and may produce instructions out of order. However, at any point in time each stage receives and produces at most one instruction. The status of the instructions flowing through the pipeline is held by a set of storage elements. Figure 4.1 shows the different pipeline stages and storage elements put together to build the microarchitecture.

At a high level, the pipeline can be divided into three parts: a front-end, a processing core, and a back-end. In the front-end (fetch/decode (FD) and rename (RN) stages), instructions are processed in program order. Program order constraints are relaxed when instructions get to the processing core (schedule/dispatch (SD), execute (EX), and recover/bypass (RB) stages). Instructions are put back into program order when they reach the back-end of the pipeline (write-back/retire (WR) stage). To explain in more detail the functionality of all the blocks that show up in figure 4.1, we describe the journey of an individual instruction I_1 from the moment it is fetched until it retires.

The journey begins when the program counter (PC) points to the location of instruction I_1 in the instruction memory[†] (IMEM). This is when the fetch/decode (FD) stage starts fetching instruction I_1 from IMEM. After fetching, the FD stage decodes instruction I_1 into its main components (*i.e.*, operation code, source operand(s), destination, *etc*). Based on these components, the FD stage computes (maybe through prediction with some types of instructions, *e.g.*, when I_1 is a conditional branch) the address of the next instruction to

[†]More accurately, we should refer to this as the instruction cache instead. However, the differences between the levels in the memory hierarchy are abstracted away here for simplification.

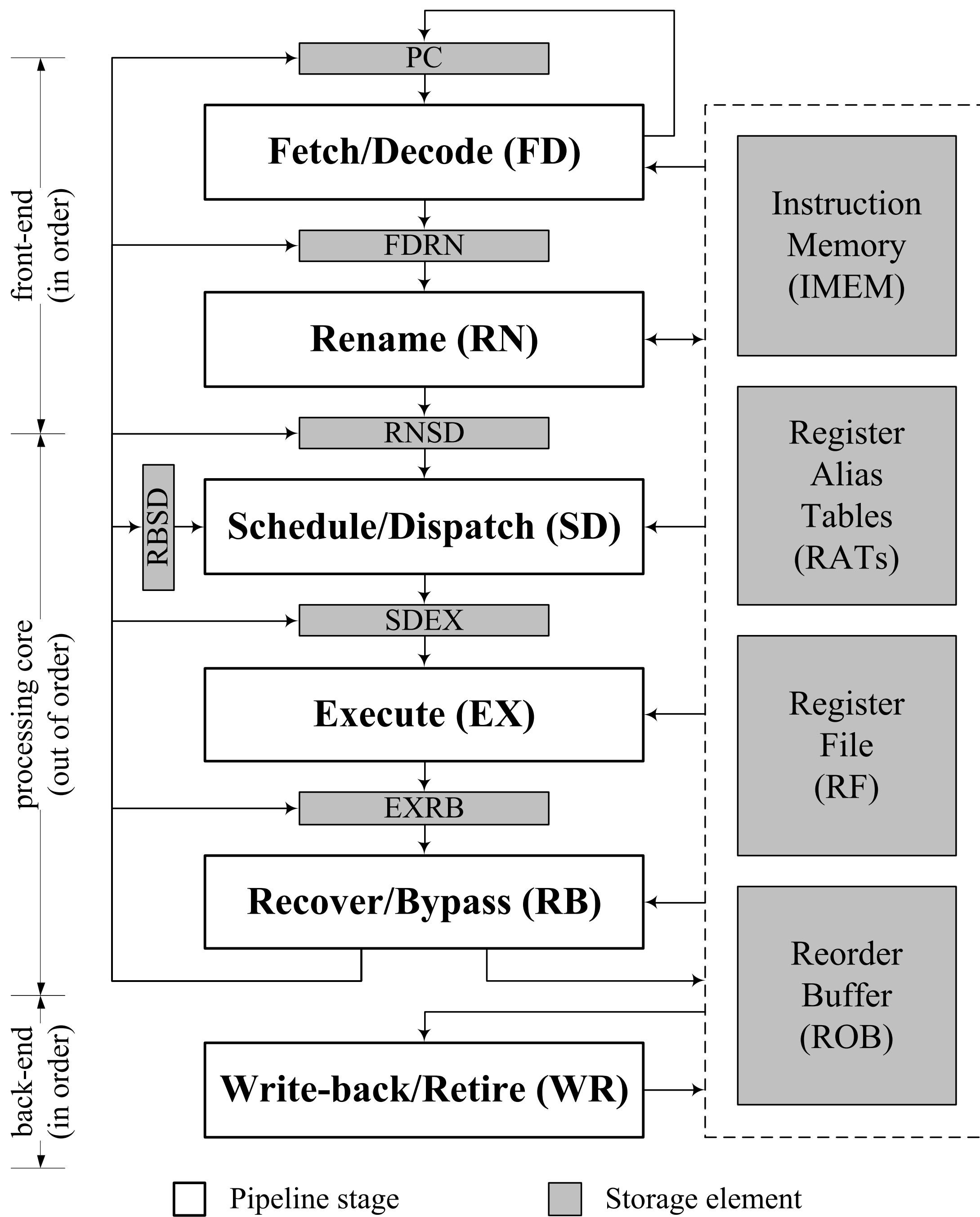

Figure 4.1: An out-of-order speculative microarchitecture

be fetched and updates the PC accordingly. At the end of this phase, the FD stage passes the components of I_1 to the rename (RN) stage through pipeline register FDRN.

The RN stage starts by assigning I_1 a unique entry at the tail of the reorder buffer (ROB). This is the location where most of the processing history of I_1 is usually recorded. Moreover, the index of that entry is used to identify I_1 throughout the rest of its journey. This index also reflects the relative age of I_1 according to program order. The latter information is needed to determine the time at which I_1 is allowed to retire; normally, instructions retire in fetch order (*i.e.*, program order).

The RN stage uses the main register alias table (RAT) in mapping the indices of the source registers of I_1 to their most recent producing instructions identified by their ROB entries; this is what is known as *register renaming*. The main RAT is updated to reflect that I_1 is now the most recent instruction that writes to the destination register. The RN stage might need to save a snapshot of the main RAT (after update) to one of the unused RATs. Creating a backup copy of the main RAT at this point helps the pipeline later recover if the instructions processed after I_1 turn out to be mispredicted. The renaming phase ends by passing I_1 to the schedule/dispatch (SD) stage through pipeline register RNSD.

The SD stage keeps I_1 in internal storage (typically known as a *reservation station*) until all its source operands are ready. After that, I_1 gets scheduled for later execution based on the availability of the appropriate execution unit. At the time of dispatch, all the source operands of I_1 need to be read. The location from which a source operand is read depends on whether the producing instruction is in-flight. As long as the producing instruction is still in-flight, the value can be obtained from the ROB entry of the producing parcel (*forwarding*) or the bypass register RBSD (*bypassing*) if possible. Otherwise, the value is obtained from the register file (RF). After reading the source operands, I_1 is dispatched for execution by moving to pipeline register SDEX.

The EX stage carries out the operation specified by I_1 and computes the result. This result is then sent along with I_1 to the recover/bypass (RB) stage through pipeline register EXRB. The RB stage then uses I_1's result in confirming the predictions that may have been made during the fetching phase of I_1. If a misprediction is detected, the recovery

takes place in three steps. First, the pipeline front-end is resteered by loading the PC with the correct address of the instruction that follows I_1. Second, the backup copy of the RAT associated with I_1 is restored. Last, all the instructions that are fetched after I_1 are invalidated. At the end of this phase, the result of I_1 is saved to its ROB entry and copied to the bypass register RBSD. By doing so, the result immediately becomes available to all the consuming instructions through forwarding or bypassing.

By writing its result to the ROB, I_1 transfers to its final processing phase during which it waits for all preceding instructions to complete. It is only after that when I_1 is permitted to retire by the write-back/retire (WR) stage. The result of I_1 is then committed to the RF and all the pipeline resources allocated to I_1, *e.g.*, ROB entry and backup RAT, are freed.

4.3 Implementing the Microarchitecture

In this section we introduce the processor model verified in our case study. The model implements the microarchitecture presented in section 4.2. However it abstracts away some of the details that may be irrelevant in verifying instruction dependencies. For instance structural hazards cannot occur in the model because it uses unbounded storage elements. Also the model uses a simplified instruction format where each instruction has only one source operand. Instruction operation codes are grouped into two abstract types: branch and arithmetic. Also the model uses uninterpreted functions in representing some of the irrelevant hardware modules such as the functional units and the decoding logic.

For the purpose of this thesis, we elaborate on how the storage elements are represented using the modeling language of Tahrir. Figure 4.2 illustrates the structure of each storage element. Most of the datatypes used here are native datatypes (see section 4.1 for details). The only exceptions are the two enumerated datatypes: `OP` and `PH`. `OP` represents the different types of instructions, in this case we have only two abstract operation codes: `"BR"` (branch) and `"AR"` (arithmetic). `PH` represents the different processing phases, *i.e.*, `"FD"`, `"RN"`, *etc.*

The PC is modeled as a variable of type `TERM`. The bypass register (RBSD) and the

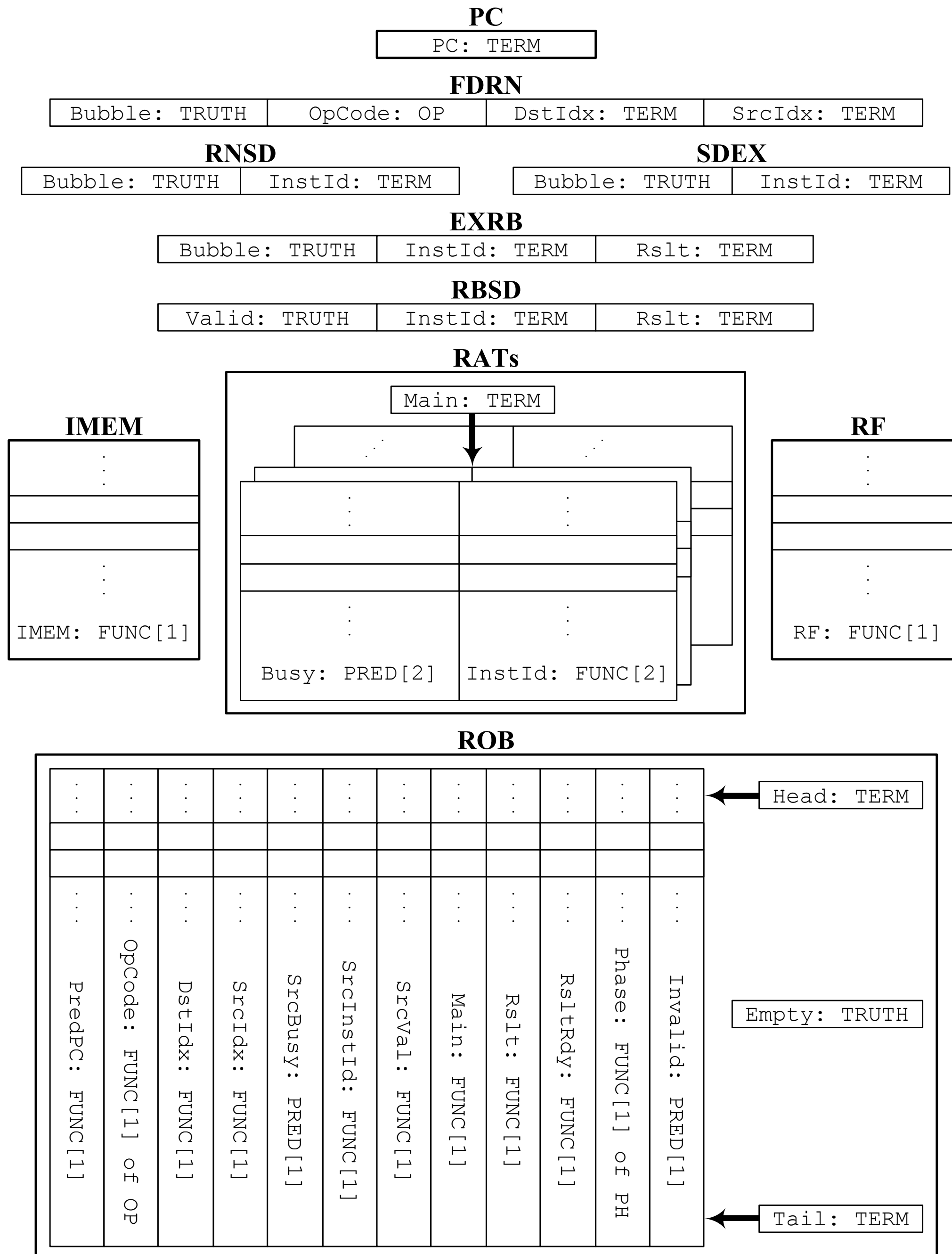

Figure 4.2: Storage elements of the processor model

pipeline registers (FDRN, RNSD, SDEX, and EXRB) are each composed of several fields which carry relevant information about the instruction held by the register. The field `Bubble`, which can be found in every pipeline register, indicates whether the register currently holds a bubble (*i.e.*, does not hold not an instruction). The other fields in the FDRN register are aimed at holding the instruction components resulted from decoding. The field `InstID`, found in the rest of the pipeline registers as well as in the bypass register, is used to carry the instruction identifier, which is also a pointer to the ROB entry associated with the instruction. The execution results of an instruction exiting the EX stage is saved to the field `Rslt` in both the EXRB and RBSD registers. The field `Valid` in the RBSD register is initialized to false and becomes true once a result is copied to the register.

IMEM is modeled as a one-argument function (`IMEM`) that maps each address to one of the program instructions. The RF is similarly modeled as a one-argument function (`RF`), yet in this case `RF` is a map from the identifiers (indices) of the architectural registers to the data held by these registers. The RATs are built out of three components: `Busy`, `InstID`, and `Main`. Each component takes two arguments. The first argument selects one of the RATs. The second argument is an index within that RAT. `Busy(i,j)` decides whether, according to the ith RAT, there exists an instruction among those in-flight that writes to the architectural register j. `InstID(i,j)` identifies the in-flight instruction behind the most recent write to the architectural register j (according to the ith RAT). The third component (`Main`) is a pointer to the main RAT, the one used during the register renaming phase.

The last storage element illustrated in figure 4.2 is the ROB. The ROB is implemented as a typical First-In-First-Out (FIFO) queue. Instructions enter the ROB from one end (pointed to by the tail pointer `Tail`) and exit from the other end (pointed to by the head pointer `Head`). Both the tail and head pointers are initialized with the index of the first ROB entry (`FirstID`). `Tail` is incremented each time an instruction enters the ROB while `Head` is incremented whenever an instruction exits the ROB. `Empty` is a flag set to true if and only if none of the ROB entries is currently occupied by any instruction.

Every ROB entry has twelve fields each of which holds history information about the associated instruction. `PredPC` is assigned the value of the PC when the instruction enters the RN stage. At the same time, the instruction components resulting from decoding are saved to `OpCode`, `DstIdx`, and `SrcIdx`. The information extracted from the RAT when

renaming the source register is kept in `SrcBusy` and `SrcInstID`. The identifier of the RAT used for renaming the instruction operands is saved to `Main`. `SrcVal` stores the value of the source operand after dispatch. After execution, instruction result is saved to `Rslt`, and `RsltRdy` is set to true. `Phase` indicates the current processing phase of the instruction. Finally, `Invalid` is a flag set when the instruction is canceled (invalidated).

The storage elements explained above are initialized such that the only instruction processed by the model is the one currently being fetched. This means that, firstly, none of the ROB entries are occupied by any instructions, *i.e.*, `ROB.Head` and `ROB.Tail` are equal, and `ROB.Empty` is true. Secondly, none of the pipeline registers holds any instructions, *i.e.*, none of the following variables is true: `FDRN.Bubble`, `RNSD.Bubble`, `SDEX.Bubble`, and `EXRB.Bubble`. Thirdly, none of the entries of the main RAT is marked busy, *i.e.*, for every integer x, $\texttt{RATs.Busy}(\texttt{RATs.Main}, \texttt{x})$ must not hold. Lastly, the value kept in the bypass register is marked invalid, *i.e.*, `RBSD.Valid` is false. All other storage elements are initially assigned non-deterministic (arbitrary) values.

4.4 Verifying Inter-instruction Dependencies

In this section, we show how we verify whether the processor model explained in section 4.3 satisfies the inter-parcel dependency obligations and consistency conditions presented in section 3.7. As mentioned before, a mechanism for identifying parcels in the model is required to be able to use the inter-parcel dependency properties. This mechanism has to include an infinite set of parcel identifiers P and a total order $\prec$ defined over the set such that it reflects the age of parcels. Since parcels are instructions in this context, and instructions are identified in the model using ROB indices, we define P to be the set of ROB indices, *i.e.*, $\{\texttt{y} \mid \texttt{y} \in \mathbb{Z} \land \texttt{y} \geq \texttt{FirstID}\}$. The less-than relation $<$ over integers is used as a total order over parcel identifiers since ROB indices are integer numbers reflecting the order at which instructions are fetched.

In order to instantiate the inter-parcel dependency properties for the processor model, we need to define the set of architectural variables and identify which physical variables are architecturally visible. A physical variable is architecturally visible if and only if it

is used to exchange values representing architectural variables between parcels. Since the register file and the program counter are the only two architectural elements in the model, the set of architectural variables V_a is defined to be $\{\texttt{PC}\} \cup \{\texttt{RF}_\texttt{x} \mid \texttt{x} \in \mathbb{Z}\}$.

The physical variables which are architecturally visible are identified by inspecting the model and determining which variables are mapped to the architectural variables and used for inter-parcel communications. The architectural program counter and register file are explicitly represented in the model by the two storage elements PC and RF respectively. Further inspection of the model reveals that instructions can exchange (read/write) the values of their operands through EXSD.Rslt and ROB.Rslt as well as RF. Therefore, we define the set of physical variables which are architecturally visible V_i^a to be $\{\texttt{RF}_\texttt{x} \mid \texttt{x} \in \mathbb{Z}\} \cup \{\texttt{PC}, \texttt{EXSD}\} \cup \{\texttt{ROB}_\texttt{y} \mid \texttt{y} \in \mathbb{Z} \wedge \texttt{y} \geq \texttt{FirstID}\}$.

To verify that the model satisfies the inter-parcel dependency properties, we check all the possible instances of each property generated using the two sets V_i^a and V_a. For example, to verify obligation **Ob1a**, we check four instances of **Ob1a** in which the two bound variables (v_a, v_i) are substituted by $(\texttt{PC}, \texttt{PC})$, $(\texttt{RF}_\texttt{x}, \texttt{RF}_\texttt{x})$, $(\texttt{RF}_\texttt{x}, \texttt{ROB}_\texttt{y})$, or $(\texttt{RF}_\texttt{x}, \texttt{EXSD})$ for arbitrary values of x and y. We refer to those four instances as $\mathbf{Ob1a}_{\texttt{PC}}^{\texttt{PC}}$, $\mathbf{Ob1a}_{\texttt{RF}_\texttt{x}}^{\texttt{RF}_\texttt{x}}$, $\mathbf{Ob1a}_{\texttt{ROB}_\texttt{y}}^{\texttt{RF}_\texttt{x}}$, and $\mathbf{Ob1a}_{\texttt{EXSD}}^{\texttt{RF}_\texttt{x}}$, respectively.

As another example, to verify the consistency condition **Cn15**, we check the two instances $\mathbf{Cn15}^{\texttt{PC}}$ and $\mathbf{Cn15}^{\texttt{RF}_\texttt{x}}$ where the variable v_a is replaced by PC and $\texttt{RF}_\texttt{x}$ respectively. Since the only possible value for the variable v_i is PC, the existential quantification can be removed and v_i can be simply substituted by PC. In other words, $\mathbf{Cn15}^{\texttt{PC}}$ can be rewritten as: $\mathbf{G}\ (\mathrm{Rd}^{\texttt{PC}}\ p \Longrightarrow \mathrm{Rd}_{\texttt{PC}}^{\texttt{PC}}\ p)$. Similarly, $\mathbf{Cn15}^{\texttt{RF}_\texttt{x}}$ can be rewritten as: $\mathbf{G}\ (\mathrm{Rd}^{\texttt{RF}_\texttt{x}}\ p \Longrightarrow \mathrm{Rd}_{\texttt{RF}_\texttt{x}}^{\texttt{RF}_\texttt{x}}\ p \vee \mathrm{Rd}_{\texttt{EXSD}}^{\texttt{RF}_\texttt{x}}\ p \vee \mathrm{Rd}_{\texttt{ROB}_\texttt{y}}^{\texttt{RF}_\texttt{x}}\ p)$, since in this case v_i shall represent one of the physical variables that are mapped to $\texttt{RF}_\texttt{x}$.

In the rest of this section, we explain how the instrumentation predicates are defined in terms of the state variables of the model (subsection 4.4.1). We also describe the lemmas we need and the assumptions we make in order to carry out the verification inductively (subsection 4.4.2).

4.4.1 Instrumentation Predicates

Most of the information needed for defining the instrumentation predicates is directly extracted from the model. In the few cases where the required information is missing, the state of the model is augmented using history variables. For instance, the model does not keep track of which instruction has recently updated the program counter. Since this information is needed to define some predicates such as $\mathrm{MRWr}_{\mathtt{PC}}$ and $\mathrm{PWr}_{\mathtt{PC}}$, we add a history variable `Hist.PCWriter` (of type `TERM`) to the model in order to memorize that piece of information. Variable `Hist.PCWriter` is initialized to some constant value `NotID` and updated with the ID of an instruction when it writes to the program counter. The constant value `NotID` should not match any of the instruction IDs. In other words, `NotID` must be outside the range of ROB indices, *i.e.*, `NotID` must be less than `FirstID`.

Similarly, we add a history variable `Hist.RFWriter` (of type `FUNC[1]`) to map each entry in the register file to the ID of the instruction which has most recently written to that entry. The last history variable, namely `Hist.PrevInst` (of type `FUNC[1]`), is used to identify for any given instruction which instruction precedes it in program order.

Before defining the instrumentation predicates, we introduce a few shortcuts to help make the predicate definitions more readable. Each shortcut is a combinational variable (of type `TRUTH`) defined in terms of the state variables of the model. Variable `Shct.Mispred` is set to true if and only if a misprediction is detected by the instruction exiting the EX unit. Variables `Shct.Alloc` and `Shct.Dealloc` mark the states at which an entry in the ROB is either allocated or released respectively. Variable `Shct.Full` becomes true if and only if all the entries in the ROB are occupied. Variable `Shct.XFull` reflects whether the ROB is about to become fully occupied in the next state (*i.e.*, whether `Shct.Full` is true in the next state).

$$\begin{aligned}\mathtt{Shct.Mispred} \equiv\ & \neg\mathtt{ROB.Invalid}(\mathtt{EXRB.InstID}) \\ & \wedge \neg\mathtt{EXRB.Bubble} \\ & \wedge \mathtt{ROB.OpCode}(\mathtt{EXRB.InstID}) = \text{"BR"} \\ & \wedge \neg(\mathtt{ROB.PredPC}(\mathtt{EXRB.InstID}) = \mathtt{EXRB.Rslt})\end{aligned}$$

$$\mathtt{Shct.Alloc} \equiv \neg\mathtt{FDRN.Bubble} \wedge \neg\mathtt{Shct.Mispred}$$

$$\mathtt{Shct.Dealloc} \equiv \neg\mathtt{ROB.Empty} \wedge \mathtt{ROB.RsltRdy}(\mathtt{ROB.Head})$$

$$\texttt{Shct.Full} \equiv \neg\texttt{ROB.Empty} \wedge \texttt{ROB.Head} = \texttt{ROB.Tail}$$

$$\begin{aligned}\texttt{Shct.XFull} \equiv\ & (\texttt{ROB.Head} = \texttt{ROB.Tail} + 1 \wedge \texttt{Shct.Alloc} \wedge \neg\texttt{Shct.Dealloc}) \\ & \vee (\texttt{Shct.Full} \wedge (\texttt{Shct.Dealloc} \Longleftrightarrow \texttt{Shct.Alloc}))\end{aligned}$$

In the remaining part of this subsection (4.4.1), we explain how the instrumentation predicates are defined for the processor model considered in our case study. We classify the predicates into three categories. The first category contains the phase predicates. Under the second category, we include all the predicates related to the architectural program counter and it physical version. The third category contains all the predicates related to the architectural register file and its physical representation in the model (*i.e.*, RF, result field of the ROB, and EXSD register).

Phase Predicates

The instrumentation predicates are meant to capture some milestones in the lifetime of a parcel. Defining those predicates for a given pipeline requires an understanding of how a parcel interacts with the pipeline state variables (*i.e.*, storage elements or physical variables) during its journey through the pipeline.

Determining the set of parcels which are in-flight at any given moment is a key in defining the phase predicates. For the processor model under consideration, those parcels are the instructions which are being processed by the different pipeline stages. At any given state, two of the in-flight instructions are processed in the front-end of the processor. These two instructions are identified by the indices `ROB.tail` and $\texttt{ROB.tail} + 1$ since these indices refer to the two ROB entries that will be eventually allocated to the instructions once they pass the front-end. The rest of the in-flight instructions (processed either at the core or the back-end) can be identified by the indices of the ROB entries which are busy. These indices range from `ROB.head` up to $\texttt{ROB.tail} - 1$.

A parcel p is in-flight (*i.e.*, Infl `p` holds) if and only if p is an index within `ROB.head` and $\texttt{ROB.tail} + 1$. ROB indices greater than $\texttt{ROB.tail} + 1$ represent parcels in the top phase since these indices refer to the ROB entries which will eventually be used by instructions yet to be processed. On the other hand, the indices of the ROB which are less than `ROB.Head`

represent parcels that exited the pipeline (*i.e.*, instructions that finished execution). The final phase of each of these parcels can be determined based on the value of the `Invalid` bit of the associated ROB entry. A value of true means that parcel is discarded, otherwise it is retired.

$\text{Infl}\ \mathtt{p} \equiv \mathtt{p} \geq \mathtt{ROB.Head} \land \mathtt{p} \leq \mathtt{ROB.Tail} + 1$

$\text{Top}\ \mathtt{p} \equiv \mathtt{p} > \mathtt{ROB.Tail} + 1$

$\text{Ret}\ \mathtt{p} \equiv \mathtt{p} < \mathtt{ROB.Head} \land \neg\mathtt{ROB.Invalid(p)}$

$\text{Dis}\ \mathtt{p} \equiv \mathtt{p} < \mathtt{ROB.Head} \land \mathtt{ROB.Invalid(p)}$

The processor model fetches and/or retires at most one instruction at any given state. For an instruction to be fetched (*i.e.*, a parcel to enter the pipeline), the FDRN register has to hold an instruction and no misprediction has to be signaled. The identity of the instruction being fetched is $\mathtt{ROB.Tail} + 2$, because this index refers to the ROB entry that the instruction will occupy. On the other hand, the identity of an instruction when it is about to exit the pipeline is `ROB.Head`, because this represents the oldest instruction in-flight. For an instruction to retire, it has to have its result ready and its ROB entry valid.

$\text{Fetch}\ \mathtt{p} \equiv \mathtt{p} = \mathtt{ROB.Tail} + 2 \land \neg\mathtt{FDRN.Bubble} \land \neg\mathtt{Shct.Mispred}$

$$\begin{aligned}\text{Retire}\ \mathtt{p} \equiv\ & \mathtt{p} = \mathtt{ROB.Head} \land \mathtt{p} < \mathtt{ROB.Tail} \\ & \land \mathtt{ROB.RsltRdy(p)} \land \neg\mathtt{ROB.Invalid(p)}\end{aligned}$$

Program Counter Predicates

Since each instruction needs to read the PC (at least once) to determine which instruction to be fetched next, the PC can be viewed as a source of data for every instruction. Therefore, the source predicate is defined such that $\text{Src}^{\mathtt{PC}}\ \mathtt{p}$ is true regardless of the value of `p`. Similarly, the address-map predicate $\text{AM}^{\mathtt{PC}}_{\mathtt{PC}}$ is defined to be true. This means that the physical PC (*i.e.*, the physical variable named PC) is statically mapped to the architectural PC regardless of the current state of the processor.

$\text{Src}^{\mathtt{PC}}\ \mathtt{p} \equiv \mathtt{True}$

$\text{AM}^{\texttt{PC}}_{\texttt{PC}} \equiv \texttt{True}$

The direct-dependency predicate is defined such that $\text{DDep}^{\texttt{PC}}\ \texttt{p}_1\ \texttt{p}_2$ holds if and only if three conditions are satisfied. First, the ID of the producer instruction $\texttt{p}_1$ is one of the ROB indices. Second, the PC is a source of data for the consumer instruction $\texttt{p}_2$, which trivially holds by definition of the source predicate. Third, $\texttt{p}_1$ is the instruction which directly precedes $\texttt{p}_2$. On the other hand, the no-dependency predicate is defined such that $\text{NoDep}^{\texttt{PC}}\ \texttt{p}$ holds if and only if the PC is a source for instruction $\texttt{p}$ and there is no producer-consumer relationship between $\texttt{p}$ and the instruction that precedes it. In this case p has to be the first instruction to be processed.

$$\text{DDep}^{\texttt{PC}}\ \texttt{p}_1\ \texttt{p}_2 \equiv \texttt{FirstID}{<}\ \texttt{p}_1 \wedge \text{Src}^{\texttt{PC}}\ \texttt{p}_2 \wedge \texttt{Hist.PrevInst}(\texttt{p}_2) = \texttt{p}_1$$

$$\text{NoDep}^{\texttt{PC}}\ \texttt{p} \equiv \text{Src}^{\texttt{PC}}\ \texttt{p} \wedge \neg\text{DDep}^{\texttt{PC}}\ \texttt{Hist.PrevInst}(\texttt{p})\ \texttt{p}$$

The definition of the read predicate for the physical PC ($\text{Rd}_{\texttt{PC}}$) has two cases. Each case addresses one of the two instructions processed in the front-end ($\texttt{ROB.Tail}$ and $\texttt{ROB.Tail}+1$), since these are the only instructions that would need to read the PC (to determine the next instruction to be fetched). In both cases, for the read to take place, the ROB shall not be about to become full and no misprediction shall be signaled. If the FDRN register carries a bubble then the read is done by the older instruction $\texttt{ROB.Tail}$, otherwise the read is done by $\texttt{ROB.Tail} + 1$. The definition of the read predicate for the architectural PC ($\text{Rd}^{\texttt{PC}}$) is identical to $\text{Rd}_{\texttt{PC}}$ because, as mentioned earlier, the physical PC is statically mapped to the architectural PC.

$$\begin{aligned}\text{Rd}_{\texttt{PC}}\ \texttt{p} \equiv\ & (\texttt{p} = \texttt{ROB.Tail} \wedge \neg\texttt{Shct.Mispred} \\ & \wedge \neg\texttt{Shct.XFull} \wedge \texttt{FDRN.Bubble}) \\ \vee\ & (\texttt{p} = \texttt{ROB.Tail} + 1 \wedge \neg\texttt{Shct.Mispred} \\ & \wedge \neg\texttt{Shct.XFull} \wedge \neg\texttt{FDRN.Bubble})\end{aligned}$$

$$\text{Rd}^{\texttt{PC}}\ \texttt{p} \equiv \text{Rd}_{\texttt{PC}}\ \texttt{p}$$

The definition of the write predicate for the physical PC ($\text{Wr}_{\texttt{PC}}$) is similar to that of the read predicate with the exception that it adds an extra case. This extra case addresses the corrective write which happens when a misprediction is detected. In this case the write is done by the instruction which has just exited the EX stage and hence identified by the

value of $\texttt{EXRB.InstID}$. The write predicate for the architectural PC ($\mathrm{Wr}^{\texttt{PC}}$) is defined to be equal to $\mathrm{Wr}_{\texttt{PC}}$ for the same reason mentioned earlier in explaining the definition of $\mathrm{Rd}^{\texttt{PC}}$. The misprediction predicate is defined such that $\mathrm{Mp}^{\texttt{PC}}\ \texttt{p}$ holds if and only if $\texttt{p}$ is the ID of the instruction held by the EXRB register and $\texttt{p}$ writes to the PC.

$$\begin{aligned}
\mathrm{Wr}_{\texttt{PC}}\ \texttt{p} \ \equiv\ & (\texttt{p} = \texttt{ROB.Tail} \wedge \neg\texttt{Shct.Mispred} \\
& \quad \wedge \neg\texttt{Shct.XFull} \wedge \texttt{FDRN.Bubble}) \\
& \vee (\texttt{p} = \texttt{ROB.Tail} + 1 \wedge \neg\texttt{Shct.Mispred} \\
& \quad \wedge \neg\texttt{Shct.XFull} \wedge \neg\texttt{FDRN.Bubble}) \\
& \vee (\texttt{p} = \texttt{EXRB.InstID} \wedge \texttt{Shct.Mispred})
\end{aligned}$$

$$\mathrm{Wr}^{\texttt{PC}}\ \texttt{p} \ \equiv\ \mathrm{Wr}_{\texttt{PC}}\ \texttt{p}$$

$$\mathrm{Wr}_{\texttt{PC}} \ \equiv\ \mathrm{Wr}_{\texttt{PC}}\ \texttt{ROB.Tail} \vee \mathrm{Wr}_{\texttt{PC}}\ \texttt{ROB.Tail} + 1 \vee \mathrm{Wr}_{\texttt{PC}}\ \texttt{EXRB.InstID}$$

$$\mathrm{Mp}^{\texttt{PC}}\ \texttt{p} \ \equiv\ \neg\texttt{EXRB.Bubble} \wedge \texttt{p} = \texttt{EXRB.InstID} \wedge \mathrm{Wr}_{\texttt{PC}}\ \texttt{p}$$

The predicate which captures the most recent write to the physical PC can be defined such that $\mathrm{MRWr}_{\texttt{PC}}\ \texttt{p}$ is true if and only if $\texttt{p}$ is the value recorded in the history variable $\texttt{Hist.PCWriter}$ and $\texttt{p}$ belongs to the set of ROB indices. The past-write predicate ($\mathrm{PWr}_{\texttt{PC}}$) can be defined on top of the most-recent-write predicate by fixing $\texttt{p}$ to the value of $\texttt{Hist.PCWriter}$. This means that $\mathrm{PWr}_{\texttt{PC}}$ holds if and only if the value of the history variable $\texttt{Hist.PCWriter}$ equals one of the ROB indices.

$$\mathrm{MRWr}_{\texttt{PC}}\ \texttt{p} \ \equiv\ \texttt{p} \geq \texttt{FirstID} \wedge \texttt{p} = \texttt{Hist.PCWriter}$$

$$\mathrm{PWr}_{\texttt{PC}} \ \equiv\ \mathrm{MRWr}_{\texttt{PC}}\ \texttt{Hist.PCWriter}$$

Register File Predicates

Each entry in the physical register file is statically mapped to the corresponding architectural register. Therefore, the address-map predicate for the physical register file ($\mathrm{AM}_{\texttt{RF}_\texttt{z}}^{\texttt{RF}_\texttt{x}}$) is defined to hold if and only if $\texttt{z}$ is the same as $\texttt{x}$. On the other hand, the ROB entries and the EXSD register are dynamically mapped to the architectural registers. A ROB entry $\texttt{ROB}_\texttt{j}$ is mapped to an architectural register $\texttt{RF}_\texttt{z}$ if and only if the destination index of instruction $\texttt{j}$ is $\texttt{z}$. Similarly, the EXSD register is mapped to $\texttt{RF}_\texttt{z}$ if and only if the destination index

of the instruction whose result is held in EXSD is $\mathtt{z}$. The mispredict predicate ($\mathrm{Mp}^{\mathtt{RF_z}}$) is defined to be false because the processor does not support executing instructions using speculative values for their operands (*i.e.*, data speculation or value prediction).

$$\mathrm{AM}^{\mathtt{RF_x}}_{\mathtt{RF_z}} \equiv \mathtt{z} = \mathtt{x}$$

$$\mathrm{AM}^{\mathtt{RF_z}}_{\mathtt{ROB_j}} \equiv \mathtt{z} = \mathtt{ROB.DstIdx(j)}$$

$$\mathrm{AM}^{\mathtt{RF_z}}_{\mathtt{EXSD}} \equiv \mathtt{z} = \mathtt{ROB.DstIdx(EXSD.InstID)}$$

$$\mathrm{Mp}^{\mathtt{RF_z}} \equiv \mathtt{False}$$

Realizing that instructions read source their operands as they exit the SD stage is key in defining the read predicates $\mathrm{Rd}_{\mathtt{RF_z}}$, $\mathrm{Rd}_{\mathtt{ROB_j}}$, and $\mathrm{Rd}_{\mathtt{EXSD}}$. More precisely, for each of these predicates to hold for an instruction $\mathtt{p}$, $\mathtt{p}$ has to be in the SDEX register. The status of the instruction producing the source operand of $\mathtt{p}$ determines the location of the read and as a consequence which one of the three predicates is true. If the producer is no longer in-flight, the value of the source operand is obtained from the RF, and hence $\mathrm{Rd}_{\mathtt{RF_z}}\ \mathtt{p}$ is true. Otherwise, $\mathrm{Rd}_{\mathtt{EXSD}}\ \mathtt{p}$ is true if the source operand is available in the SDEX register, else, the source operand shall be read from the ROB and so $\mathrm{Rd}_{\mathtt{ROB_z}}\ \mathtt{p}$ is true. The read predicate for architectural registers is simply defined such that $\mathrm{Rd}^{\mathtt{RF_z}}\ \mathtt{p}$ holds if and only if some physical location mapped to architectural register $\mathtt{RF_z}$ is read by instruction $\mathtt{p}$.

$$\begin{aligned}\mathrm{Rd}_{\mathtt{RF_z}}\ \mathtt{p} \equiv\ & \neg\mathtt{SDEX.Bubble} \\ & \wedge \mathtt{p} = \mathtt{SDEX.InstID} \wedge \mathtt{z} = \mathtt{ROB.SrcIdx(p)} \\ & \wedge \neg(\mathtt{ROB.SrcBusy(p)} \\ & \qquad \wedge \mathtt{ROB.SrcInstID(p)} \geq \mathtt{ROB.Head} \\ & \qquad \wedge \mathtt{ROB.SrcInstID(p)} < \mathtt{ROB.Tail})\end{aligned}$$

$$\begin{aligned}\mathrm{Rd}_{\mathtt{ROB_j}}\ \mathtt{p} \equiv\ & \neg\mathtt{SDEX.Bubble} \\ & \wedge \mathtt{p} = \mathtt{SDEX.InstID} \wedge \mathtt{j} = \mathtt{ROB.SrcInstID(p)} \\ & \wedge (\mathtt{ROB.SrcBusy(p)} \\ & \qquad \wedge \mathtt{ROB.SrcInstID(p)} \geq \mathtt{ROB.Head} \\ & \qquad \wedge \mathtt{ROB.SrcInstID(p)} < \mathtt{ROB.Tail}) \\ & \wedge (\mathtt{EXSD.Bubble} \vee \mathtt{EXSD.InstID} \neq \mathtt{ROB.SrcInstID(p)})\end{aligned}$$

$$\begin{aligned}\mathrm{Rd}_{\mathtt{EXSD}}\ \mathtt{p}\ \equiv\ & \neg\mathtt{SDEX.Bubble} \\ & \wedge \mathtt{p} = \mathtt{SDEX.InstID} \\ & \wedge (\mathtt{ROB.SrcBusy(p)} \\ & \quad \wedge \mathtt{ROB.SrcInstID(p)} \geq \mathtt{ROB.Head} \\ & \quad \wedge \mathtt{ROB.SrcInstID(p)} < \mathtt{ROB.Tail}) \\ & \wedge \neg\mathtt{EXSD.Bubble} \wedge \mathtt{EXSD.InstID} = \mathtt{ROB.SrcInstID(p)}\end{aligned}$$

$$\begin{aligned}\mathrm{Rd}^{\mathtt{RF_z}}\ \mathtt{p}\ \equiv\ & \mathrm{Rd}_{\mathtt{RF_z}}\ \mathtt{p} \\ & \vee \mathrm{Rd}_{\mathtt{ROB_{ROB.SrcInstID(p)}}}\ \mathtt{p} \wedge \mathrm{AM}^{\mathtt{RF_z}}_{\mathtt{ROB_{ROB.SrcInstID(p)}}} \\ & \vee \mathrm{Rd}_{\mathtt{EXSD}}\ \mathtt{p} \wedge \mathrm{AM}^{\mathtt{RF_z}}_{\mathtt{EXSD}}\end{aligned}$$

There are two moments at which an instruction $\mathtt{p}$ writes a value to a physical location mapped to its destination register. First, when $\mathtt{p}$ finishes execution, its result is written to both its ROB entry $\mathtt{ROB_j}$ (where $\mathtt{p}$ equals $\mathtt{j}$) and the EXSD register. In this case both $\mathrm{Wr}_{\mathtt{ROB_j}}\ \mathtt{p}$ and $\mathrm{Wr}_{\mathtt{EXSD}}\ \mathtt{p}$ shall be true. Second, when p retires, its result is written to the RF entry $\mathtt{RF_z}$ associated with its destination unless p has been invalidated. In this case $\mathrm{Wr}_{\mathtt{RF_z}}\ \mathtt{p}$ must be true. The write predicate for architectural registers is defined, in analogy with the corresponding read predicate, such that $\mathrm{Wr}^{\mathtt{RF_z}}\ \mathtt{p}$ holds if and only if p writes its result to any of the physical locations mapped to architectural register $\mathtt{RF_z}$.

$$\begin{aligned}\mathrm{Wr}_{\mathtt{RF_z}}\ \mathtt{p}\ \equiv\ & \neg\mathtt{ROB.Empty} \wedge \mathtt{ROB.RsltRdy(p)} \wedge \neg\mathtt{ROB.Invalid(p)} \\ & \wedge \mathtt{p} = \mathtt{ROB.Head} \wedge \mathtt{z} = \mathtt{ROB.DstIdx(p)}\end{aligned}$$

$$\mathrm{Wr}_{\mathtt{ROB_j}}\ \mathtt{p}\ \equiv\ \neg\mathtt{EXRB.Bubble} \wedge \mathtt{p} = \mathtt{EXRB.InstID} \wedge \mathtt{j} = \mathtt{p}$$

$$\mathrm{Wr}_{\mathtt{EXSD}}\ \mathtt{p}\ \equiv\ \neg\mathtt{EXRB.Bubble} \wedge \mathtt{p} = \mathtt{EXRB.InstID}$$

$$\begin{aligned}\mathrm{Wr}^{\mathtt{RF_z}}\ \mathtt{p}\ \equiv\ & \mathrm{Wr}_{\mathtt{RF_z}}\ \mathtt{p} \\ & \vee \mathrm{Wr}_{\mathtt{ROB_p}}\ \mathtt{p} \wedge \mathrm{AM}^{\mathtt{RF_z}}_{\mathtt{ROB_p}}\end{aligned}$$

The predicates which capture the anonymous writes to the three physical representations of the architectural registers can be expressed in terms of the write predicates defined above. For instance, the predicate $\mathrm{Wr}_{\mathtt{RF_z}}$ is defined to be true if and only if the instruction at the head of the ROB writes to $\mathtt{RF_z}$. On the other hand, the predicates $\mathrm{Wr}_{\mathtt{ROB_j}}$ and $\mathrm{Wr}_{\mathtt{EXSD}}$ hold if and only if the instruction kept in the EXRB register writes to $\mathtt{ROB_j}$ and $\mathtt{EXRB}$ respectively.

$$\mathrm{Wr}_{\mathtt{RF_z}} \equiv \mathrm{Wr}_{\mathtt{RF_z}}\ \mathtt{ROB.Head}$$

$$\mathrm{Wr}_{\mathtt{ROB_j}} \equiv \mathrm{Wr}_{\mathtt{ROB_j}}\ \mathtt{EXRB.InstID}$$

$$\mathrm{Wr}_{\mathtt{EXSD}} \equiv \mathrm{Wr}_{\mathtt{EXSD}}\ \mathtt{EXRB.InstID}$$

The most-recent-write predicate associated with the RF is defined such that $\mathrm{MRWr}_{\mathtt{RF_z}}\ \mathtt{p}$ is true if and only if $\mathtt{p}$ is the instruction which makes the most recent write to $\mathtt{RF_z}$ and $\mathtt{p}$ is a ROB index. The predicate associated with the ROB is defined such that $\mathrm{MRWr}_{\mathtt{ROB_j}}\ \mathtt{p}$ holds if instruction $\mathtt{p}$ occupies the entry $\mathtt{ROB_j}$ and the result saved at that entry is ready. For the EXSD, $\mathrm{MRWr}_{\mathtt{EXSD}}\ \mathtt{p}$ shall hold if and only if $\mathtt{p}$ is the instruction whose result is in the EXSD register and $\mathtt{p}$ is one of the ROB indices.

$$\mathrm{MRWr}_{\mathtt{RF_z}}\ \mathtt{p} \equiv \mathtt{p} = \mathtt{Hist.RFWriter(z)} \wedge \mathtt{p} \geq \mathtt{FirstID}$$

$$\mathrm{MRWr}_{\mathtt{ROB_j}}\ \mathtt{p} \equiv \mathtt{ROB.RsltRdy(p)} \wedge \mathtt{p} < \mathtt{ROB.Tail} \wedge \mathtt{p} \geq \mathtt{FirstID} \wedge \mathtt{p} = \mathtt{j}$$

$$\mathrm{MRWr}_{\mathtt{EXSD}}\ \mathtt{p} \equiv \neg\mathtt{EXSD.Bubble} \wedge \mathtt{p} = \mathtt{EXSD.InstID} \wedge \mathtt{p} \geq \mathtt{FirstID}$$

The past-write predicate for the RF is defined to be true for an entry $\mathtt{RF_z}$ if and only if the history variable RFWriter(z) contains a ROB index. The past-write predicate for the ROB and the EXSD register are defined to be true. The reason behind that is to make sure that in these two cases obligation **Ob3a** is reduced to the negation of the precedent. In other words, to satisfy **Ob3a**, an instruction cannot make its final read for the source operand from neither the ROB nor the EXSD if that instruction does not depend on any older instruction.

$$\mathrm{PWr}_{\mathtt{RF_z}} \equiv \mathtt{Hist.RFWriter(z)} \geq \mathtt{FirstID}$$

$$\mathrm{PWr}_{\mathtt{ROB_j}} \equiv \mathtt{True}$$

$$\mathrm{PWr}_{\mathtt{EXSD}} \equiv \mathtt{True}$$

The source predicate $\mathrm{Src}^{\mathtt{RF_z}}\ \mathtt{p}$ holds if and only if $\mathtt{z}$ is the index of the source operand of instruction $\mathtt{p}$. For a pair of instructions $\mathtt{p_1}$ and $\mathtt{p_2}$, the direct-dependency predicate shall be true if and only if $\mathtt{RF_z}$ is the source operand of $\mathtt{p_2}$ and $\mathtt{p_1}$ is the producer. The no-dependency predicate holds for an instruction $\mathtt{p}$ if and only if $\mathtt{RF_z}$ is the source operand of $\mathtt{p}$, none of the in-flight instructions is a producer, and no instruction has written to $\mathtt{RF_z}$.

$$\mathrm{Src}^{\mathtt{RF_z}}\ \mathtt{p} \equiv \mathtt{p} < \mathtt{ROB.Tail} \wedge \mathtt{z} = \mathtt{ROB.SrcIdx(p)}$$

$$\text{DDep}^{\texttt{RF}_\texttt{z}}\ \texttt{p}_1\ \texttt{p}_2 \equiv \text{Src}^{\texttt{RF}_\texttt{z}}\ \texttt{p}_2 \land ((\texttt{ROB.SrcBusy}(\texttt{p}_2) \land \texttt{p}_1 = \texttt{ROB.SrcInstID}(\texttt{p}_2)) \lor (\neg\texttt{ROB.SrcBusy}(\texttt{p}_2) \land \texttt{p}_1 = \texttt{Hist.RFWriter}(\texttt{z}) \land \texttt{p}_1 \geq \texttt{FirstID}))$$

$$\text{NoDep}^{\texttt{RF}_\texttt{z}}\ \texttt{p} \equiv \text{Src}^{\texttt{RF}_\texttt{z}}\ \texttt{p} \land \neg\texttt{ROB.SrcBusy}(\texttt{p}) \land \texttt{Hist.RFWriter}(\texttt{z}) < \texttt{FirstID}$$

4.4.2 Lemmas and Assumptions

In verifying the inter-parcel dependency properties, we needed to make three assumptions. One of these three assumptions states that the constant value `NotID` never matches the ID of any instruction (*i.e.*, an index in the ROB). Since ROB indices start at $FirstID$, we simply assume that `NotID` is less than `FirstID`.

The other two assumptions specify the behavior of the two black-box stages SD and EX respectively. One assumption restricts the SDEX register (which otherwise contains a non-deterministic value) to instructions that are currently in-flight. It also makes sure that the source operand of the instruction in the SDEX register is available in the RF, the EXSD register, or the ROB. The other assumption targets the EXRB register and makes sure it holds one of those in-flight instructions whose results are not yet ready.

Since Tahrir implements an inductive approach for verifying SSLTL properties, additional lemmas are typically needed to strengthen the inductive invariant, and hence, keep induction within the reachable state-space. In verifying the inter-parcel dependency properties against the processor model, we needed (to verify) a total of 43 lemmas to exclude unreachable states. These lemmas can be classified into three categories:

1. Lemmas relating the different state variables of the processor model (*i.e.*, ROB fields, pipeline registers, *etc*) to each other: a total of 19 lemmas belong to this category. Lemmas **Lm1** and **Lm16** are detailed below for illustration.

   Lm1: The ROB is not full and the head pointer is less than or equal to the tail pointer
   $\mathbf{G}\ (\neg\texttt{Shct.Full} \land \texttt{ROB.Head} \leq \texttt{ROB.Tail})$

Lm16: If $\mathtt{p}$ is an in-flight branch instruction occupying an entry in the ROB and $\mathtt{p}$ has neither produced a result nor been invalidated yet, then the ID of the RAT associated with $\mathtt{p}$ has to be less than the ID of the current RAT[‡].

$$\mathbf{G}\left(\begin{array}{l}\left(\begin{array}{l}\mathtt{p} < \mathtt{ROB.Tail} \wedge \mathtt{p} \geq \mathtt{ROB.Head} \\ \wedge \neg\mathtt{ROB.Invalid(p)} \wedge \neg\mathtt{ROB.RsltRdy(p)} \\ \wedge\ \mathtt{ROB.OpCode(p)} = \text{"BR"}\end{array}\right) \Longrightarrow \\ \mathtt{ROB.Main(p)} < \mathtt{RATs.Main}\end{array}\right)$$

2. Lemmas relating state variables of the processor model to history variables: this category includes six lemmas. Lemma **Lm20** is explained below as an example.

Lm20: If $\mathtt{p}$ identifies an in-flight instruction occupying an entry in the ROB, then the ID of the preceding instruction either belongs to the set of ROB indices or equals to the constant $\mathtt{NotID}$, and in either case that ID has to be less than $\mathtt{p}$.
$\forall\, \mathtt{p}.$

$$\mathbf{G}\left(\begin{array}{l}\mathtt{p} < \mathtt{ROB.Tail} \wedge \mathtt{p} \geq \mathtt{ROB.Head} \Longrightarrow \\ \quad \mathtt{Hist.PrevInst(p)} < \mathtt{p} \\ \quad \wedge\ (\mathtt{Hist.PrevInst(p)} \geq \mathtt{FirstID} \vee \mathtt{Hist.PrevInst(p)} = \mathtt{NotID})\end{array}\right)$$

3. Lemmas relating the states of the Büchi automata (generated from the obligations) to the state variables of the processor model: a total of 18 lemmas are classified under this category. To illustrate these lemmas, we elaborate below on lemma **Lm35** which addresses the states of the Büchi automaton generated for obligation **Ob3a**, more specifically its instance $\mathbf{Ob3a}^{\mathtt{PC}}_{\mathtt{PC}}$. The Büchi automaton for that obligation is shown in figure 4.3.

Lm35: If an instruction $\mathtt{p}$ is not in the state $\mathtt{q}_0$, $\mathtt{p}$ shall be less than or equal to the index of the ROB tail. If $\mathtt{p}$ equals the tail index, either the FDRN register does not carry an instruction or the ID of the preceding instruction is among the set of ROB indices. Otherwise, either $\mathtt{p}$ does not currently depend on any

[‡]This implies that the RAT used at the time instruction $\mathtt{p}$ was in the RN stage is a past snapshot of the current RAT.

previous instructions with respect to PC or $\mathtt{p}$ has been invalidated.
$\forall\, \mathtt{p}.$

$$\mathbf{G}\left(\begin{array}{l}\neg \mathtt{q}_0(\mathtt{p}) \Longrightarrow \\ \quad \mathtt{p} = \mathtt{ROB.Tail} \wedge (\mathtt{FDRN.Bubble} \vee \mathtt{Hist.PrevInst}(\mathtt{p}) \geq \mathtt{FirstID}) \\ \quad \vee\, \mathtt{p} < \mathtt{ROB.Tail} \wedge (\neg \mathrm{NoDep}^{\mathtt{PC}}\ \mathtt{p} \vee \mathtt{ROB.Invalid}(\mathtt{p}) \end{array}\right)$$

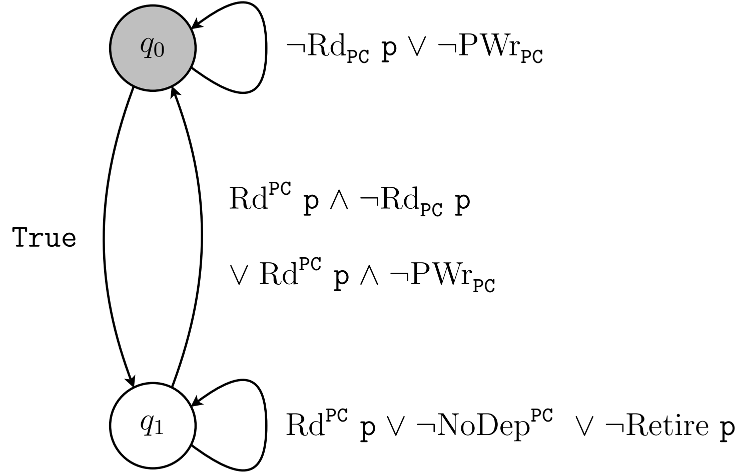

Figure 4.3: Büchi automaton for obligation $\mathbf{Ob3a}^{\mathtt{PC}}_{\mathtt{PC}}$

4.5 Verification Remarks

The total number of properties verified in our case study is 83. These properties can be divided into three categories:

- The first category contains 14 properties, all of which are instances of the inter-parcel obligations. The properties in this category are listed as follows:

$\mathbf{Ob1a}^{\mathtt{PC}}_{\mathtt{PC}}$	$\mathbf{Ob1a}^{\mathtt{RF_x}}_{\mathtt{RF_x}}$	$\mathbf{Ob1a}^{\mathtt{RF_x}}_{\mathtt{EXSD}}$	$\mathbf{Ob1a}^{\mathtt{RF_x}}_{\mathtt{ROB_y}}$
$\mathbf{Ob2}^{\mathtt{PC}}_{\mathtt{PC}}$	$\mathbf{Ob2}^{\mathtt{RF_x}}_{\mathtt{RF_x}}$	$\mathbf{Ob2}^{\mathtt{RF_x}}_{\mathtt{EXSD}}$	$\mathbf{Ob2}^{\mathtt{RF_x}}_{\mathtt{ROB_y}}$
$\mathbf{Ob3a}^{\mathtt{PC}}_{\mathtt{PC}}$	$\mathbf{Ob3a}^{\mathtt{RF_x}}_{\mathtt{RF_x}}$	$\mathbf{Ob3a}^{\mathtt{RF_x}}_{\mathtt{EXSD}}$	$\mathbf{Ob3a}^{\mathtt{RF_x}}_{\mathtt{ROB_y}}$
$\mathbf{Ob4}^{\mathtt{PC}}$	$\mathbf{Ob4}^{\mathtt{RF_x}}$	—	—

- The second category contains 26 properties. These properties represent all the instances of the inter-parcel consistency conditions. The following is a list of all the properties in this category:

$\mathbf{Cn1b}^{\mathtt{PC}}_{\mathtt{PC}}$	$\mathbf{Cn1b}^{\mathtt{RF_x}}_{\mathtt{RF_x}}$	$\mathbf{Cn1b}^{\mathtt{RF_x}}_{\mathtt{EXSD}}$	$\mathbf{Cn1b}^{\mathtt{RF_x}}_{\mathtt{ROB_y}}$
$\mathbf{Cn1c}^{\mathtt{PC}}_{\mathtt{PC}}$	$\mathbf{Cn1c}^{\mathtt{RF_x}}_{\mathtt{RF_x}}$	$\mathbf{Cn1c}^{\mathtt{RF_x}}_{\mathtt{EXSD}}$	$\mathbf{Cn1c}^{\mathtt{RF_x}}_{\mathtt{ROB_y}}$
$\mathbf{Cn3b}^{\mathtt{PC}}_{\mathtt{PC}}$	$\mathbf{Cn3b}^{\mathtt{RF_x}}_{\mathtt{RF_x}}$	$\mathbf{Cn3b}^{\mathtt{RF_x}}_{\mathtt{EXSD}}$	$\mathbf{Cn3b}^{\mathtt{RF_x}}_{\mathtt{ROB_y}}$
Cn5	—	—	—
Cn6	—	—	—
Cn7	—	—	—
Cn8	—	—	—
Cn9	—	—	—
Cn10	—	—	—
Cn11	—	—	—
Cn12	—	—	—
$\mathbf{Cn13}^{\mathtt{PC}}$	$\mathbf{Cn13}^{\mathtt{RF_x}}$	—	—
$\mathbf{Cn14}^{\mathtt{PC}}$	$\mathbf{Cn14}^{\mathtt{RF_x}}$	—	—
$\mathbf{Cn15}^{\mathtt{PC}}$	$\mathbf{Cn15}^{\mathtt{RF_x}}$	—	—

- The third contains 43 properties. These properties are the lemmas introduced for the purpose of limiting induction scope to reachable states. Examples of these lemmas can be found in subsection 4.4.2.

The machine on which we ran our verification experiments is a Linux box that has a 3.2 MHz Intel dual-core processor with 3.4 GB of memory. The cumulative CPU time taken for verifying the properties by Tahrir using the CVC3 engine is 6.27 minutes. It takes 7.25 minutes to accomplish the same task using the UCLID engine. The maximum memory consumption is 157.5 MB and 171.2 MB during the two runs respectively.

Figure 4.4 has two plots that show the CPU runtime and memory consumed in verifying each property during the two runs. The plotted data include the time and memory consumed in generating and optimizing the automata. These data also include the time and memory consumed by the decision engine (*i.e.*, CVC3 and UCLID).

Figure 4.4: CPU time and memory consumption: CVC3 (Top) and UCLID (Bottom)

In the plots shown in figure 4.4, the properties are categorized based on the number of inner quantifiers (NIQ) used in their proof statements. Based on the plotted data, properties with relatively higher NIQs tend to consume more resources during verification. This is not surprising because the inner quantifiers appear in the final formula which gets sent to the decision engine. Hence, a higher number of quantifiers would possibly lead to more variable instantiations and consequently increase the amount of resources consumed in determining the falsifiability of the formula.

4.6 Summary

Tahrir, a tool that uses an SMT solver as a decision engine, is implemented to verify SSLTL properties inductively. Tahrir is used to verify inter-instruction dependencies in a processor modeled with an abstract datapath. The processor model supports out-of-order speculative execution of instructions. The instrumentation predicates are defined in terms of the variables of the model. The inter-instruction correctness is instantiated into 14 obligations and 26 consistency conditions. The total number of properties verified is 83 (including 43 lemmas for strengthening the induction). Verification takes less than 8 minutes and consumes less 200MB of memory.

Chapter 5

Conclusions

Our thesis contributions can be summarized in two main points:

1. Introducing an inductive approach for verifying SSLTL properties: in our approach, the target SSLTL property is transformed into an invariant which can be then checked by induction using an SMT solver or an invariant checker. Our approach is shown to be sound and complete with respect to the standard definition of LTL correctness.

2. Presenting a strategy for verifying whether a pipelined microprocessor preserves data and control dependencies among instructions: in our strategy, data and control dependencies are treated uniformly. Our top-level correctness criteria are decomposed into a set of safety properties that allow flexible forms of speculative out-of-order execution of instructions.

Chapter 2 presents an algorithmic view of our approach for verifying SSLTL properties by k-step induction. The main function *Verify* takes (among other inputs) a model M and an SSLTL property p, and returns true if and only if the M satisfies p. It also takes a number k representing the depth of the induction and a Boolean expression e to use in strengthening the inductive invariant.

Function *Verify* starts by translating the property p into an automaton B_p. The translation is an implementation of the basic Büchi automata construction algorithm with the

exception that all the states of the constructed automaton are considered accepting states. The automaton B_p is transformed to a model M_p by encoding its states as Boolean variables. An invariant e_p (written in terms of these variables) is built to describe the states and transitions of B_p. The augmented model M_a is formed by adding M_p to the original model M.

Function *Verify* then checks whether the augmented model M_a satisfies the invariant e_p using k-step induction. During induction, the invariant is strengthened using e to limit unreachable states. A value of true is returned by function *Verify* if and only if the induction shows that the augmented model M_p satisfies the strengthened invariant $e_p \wedge e$ for the given values of e and k.

Our approach for SSLTL verification is proven to be sound and complete. The core of the proof is theorem 2.1 which is used to show that applying our SSLTL verification algorithm (represented by function *Verify*) on a model M and an SSLTL property is equivalent to verifying that M satisfies p.

Chapter 3 explains how we specify that a microprocessor handles dependencies between instructions correctly. In our correctness definitions, we describe the way instructions should interact with the state variables of the microprocessor as well as with each other. We present the correctness in the context of generic pipelines where instructions are referred to as parcels.

The state of the pipeline is captured by a set of physical variables. These physical variables can be statically or dynamically mapped to architectural variables. Parcels interact with the state of the pipeline by reading from and/or writing to the physical variables. A parcel can make an arbitrary number of reads from and/or writes to physical variables that are mapped to the same architectural variable. All but final reads and writes are considered speculative and hence do not affect the final results of a parcel.

A pipeline is instrumented with a parcel identification mechanism and some predicates to monitor parcel activities. Predicates such as Top and Infl probe the phase of a parcel at any given state. Predicates such as Rd and Wr mark the interactions between a parcel and the variables of the pipeline. History information about these interactions are captured by predicates such as MRWr and PWr. Different types of inter-parcel dependencies are

identified by predicates such as DDep and NoDep.

Using this instrumentation, inter-parcel correctness is expressed in the form of two properties that support speculative out-of-order processing of parcels: **PropProdCons** and **PropNoProd**. **PropProdCons** describes the interaction between any two parcels where the leading parcel produces a data to be consumed by the trailing parcel. **PropNoProd** addresses the case in which a parcel consumes a data that is not produced by any leading parcel.

Due to their temporal complexity, properties **PropProdCons** and **PropNoProd** are decomposed into 4 obligations and 14 consistency conditions. The purpose of the obligations is to detect bugs in the implementation. The consistency conditions ensure that the definitions of the instrumentation predicates are correct. The decomposition is proven to be sound.

Chapter 4 describes a case study that we conducted to illustrate our verification techniques (presented in chapters 2 and 3) and evaluate their effectiveness. Tahrir, a tool that uses an SMT solver as a decision engine, is developed to verify SSLTL properties about term-level models inductively. Tahrir implements the algorithm given by function *Verify*. The verification is guided by proof statements specifying assumptions about the model and lemmas to keep induction within reachable states.

Tahrir is used to verify inter-instruction dependencies in a processor modeled with an abstract datapath. The processor model supports out-of-order speculative execution of instructions. The architectural variables in the model are a program counter (PC) and a register file (RF). The model uses a register alias table (RAT) and a reorder buffer (ROB) to implement register renaming. The ROB is also used for bookkeeping and for exchanging (forwarding) data between in-flight instructions. Instructions can also exchange data using a bypass register (RBSD).

The parcel-based instrumentation technique is applied to the model. Parcels (instructions) are identified by ROB indices. The instrumentation predicates are defined in terms of the model variables in addition to a few history variables. The inter-instruction correctness is instantiated into 14 obligations and 26 consistency conditions. This instantiation relies on manually specifying which physical variables in the model are architecturally visible.

The total number of properties verified in our case study is 83. This number includes 43 lemmas that are introduced to strengthen the induction. Tahrir takes less than 8 minutes and consumes less than 200MB of memory in verifying the properties. Our results show some correlation between the number of inner quantifier (NIQ) variables used in the proof statement of a property and the resources consumed in verifying that property. Properties with higher NIQ tend to take longer time and/or consume more memory to verify.

In formulating the correctness criteria and decompositions, our goal was to make the criteria as general as possible. For example, in our case study, we were able to use the same correctness criteria for both the program counter and register file (*i.e.*, for both control and data dependencies). However, further evaluation on a wider range of pipelines will be necessary to validate the true generality of our criteria and strategy.

Our plan for future research is four-fold: (1) evaluate the *generality* of our inter-parcel verification strategy, (2) extend its *applicability*, (3) boost the *performance* of our SSLTL verification tool (Tahrir), and (4) enhance its *usability*.

To evaluate the generality of our inter-parcel verification strategy, we will apply our strategy to microprocessor models that support optimizations that are not covered in our case study such as exceptions and value prediction. We will also conduct case studies focused on other types of pipelined systems, *i.e.*, non-processor pipelines such as those used in image and/or video processing.

To extend its applicability, we will combine our strategy with a mechanism for automatic datapath abstraction such as the one proposed by Ciubotario [9]. This will allow us to tackle more concrete models of microprocessors, *i.e.*, bit-level models as opposed to term-level models similar to the one verified in our case study.

To boost tool's performance, we will replace the current system-call-based interface at the core of Tahrir with an API-based interface. We believe a tighter coupling between Tahrir and its decision engine would enhance overall performance. We will also implement a more efficient algorithm for generating Büchi automata such as those developed by Couvreur [14], Gastin *et al.* [17], and Latvala [37].

To enhance the tool's usability, we will provide Tahrir with a capability that helps the user come up with the invariants. To do so, we will explore the existing techniques for

automatic generation of invariants such as those presented by Bensalem *et al.* [4].

References

[1] Mark Aagaard. A Hazards-Based Correctness Statement for Pipelined Circuits. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2003. 53

[2] Mark Aagaard, Nancy Day, and Meng Lou. Relating Multi-step and Single-step Correctness Statements. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 2517 of *Lecture Notes in Computer Science*, pages 123–141. Springer, 2002. 53

[3] Clark Barrett and Cesare Tinelli. CVC3. In *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany. 39, 89

[4] Saddek Bensalem and Yassine Lakhnech. Automatic Generation of Invariants. *Formal Methods in System Design*, 15:75–92, July 1999. 118

[5] Bob Bentley. Validating a Modern Microprocessor. In *Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 2–4. Springer, 2005. 1

[6] Randal Bryant, Shuvendu Lahiri, and Sanjit Seshia. Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2002. 2, 39, 89

[7] Julius Richard Büchi. Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic. In *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science*, volume 44 of *Studies in Logic and the Foundations of Mathematics*, pages 1 – 11. Elsevier, 1966. 9

[8] Jerry Burch and David Dill. Automatic verification of Pipelined Microprocessor Control. In *Computer Aided Verification (CAV)*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer, 1994. 52

[9] Vlad Ciubotariu. *Automatic Datapath Abstraction of Pipelined Circuits*. PhD thesis, School of Computer Science, University of Waterloo, February 2011. 117

[10] Edmund Clarke, Ernest Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, April 1986. 6, 12

[11] Edmund Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10:47–71, 1997. 15

[12] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[13] Costas Courcoubetis, Moshe Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. In *Computer Aided Verification (CAV)*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242, London, UK, 1991. Springer-Verlag. 16

[14] Jean-Michel Couvreur. On-the-fly Verification of Linear Temporal Logic. In *FM99 Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 711–711. Springer Berlin / Heidelberg, 1999. 16, 117

[15] Leonardo Mendonça de Moura, Sam Owre, Harald Rueß, John M. Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In *Computer Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500. Springer, 2004. 4

[16] Dov Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In *Temporal Logic in Specification*, pages 409–448, London, UK, 1987. Springer-Verlag. 15

[17] Paul Gastin and Denis Oddoux. Fast LTL to Büchi Automata Translation. In *Computer Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, London, UK, 2001. Springer-Verlag. 16, 117

[18] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple On-the-Fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd. 16, 17, 18, 91

[19] Ganesh Gopalakrishnan. *Computation Engineering: Applied Automata Theory and Logic*. Springer-Verlag, first edition, 2006.

[20] Michael Gordon. HOL: A Proof Generating System for Higher Order Logic. In *VLSI Specification, Verification and Synthesis*, chapter 3, pages 73–128. Kluwer Academic Publishers, 1988. 2

[21] Aarti Gupta. Formal Hardware Verification Methods: A Survey. *Formal Methods in System Design*, 1(2/3):151–238, 1992. 2

[22] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003. 44, 46, 47

[23] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The Microarchitecture of the Pentium® 4 Processor. *Intel Technology Journal*, Q1, 2001.

[24] Gerard Holzmann, Doron Peled, and Mihalis Yannakakis. On Nested Depth First Search. In *Proceedings of the 2nd Spin Workshop. The Spin Verification System*, pages 23–32. American Mathematical Society, 1996. 16

[25] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Decomposing the Proof of Correctness of Pipelined Microprocessors. In *Computer Aided Verification*

(CAV), volume 1427 of *Lecture Notes in Computer Science*, pages 122–134. Springer, 1998. 52

[26] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Proof of Correctness of a Processor with Reorder Buffer using the Completion Functions Approach. In *Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*, pages 47–59. Springer, 1999. 52

[27] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

[28] Robert Jones, Jens Skakkebæk, and David Dill. Reducing Manual Abstraction in Formal Verification of Out-of-Order Execution. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1522 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 1998.

[29] Gerry Kane and Joseph Heinrich. *MIPS RISC Architecture*. Prentice-Hall, second edition, 1992. 49

[30] Matt Kaufmann and J Strother Moore. Design Goals for ACL2. Technical Report 101, Computational Logic, Inc., August 1994. 2

[31] Matt Kaufmann and J Strother Moore. ACL2: An Industrial Strength Version of NQTHM. In *Eleventh Annual Conference on Computer Assurance (COMPASS)*, pages 23–34. IEEE Computer Society Press, 1996. 2

[32] Christoph Kern and Mark Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999. 2

[33] Peter Kogge. *The Architecture of Pipelined Computers*. McGraw Hill, 1981. 44

[34] Dexter Kozen. Results on the Propositional μ-Calculus. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*, pages 348–359, London, UK, 1982. Springer-Verlag. 12

[35] Orna Kupferman and Moshe Vardi. Model Checking of Safety Properties. *Formal Methods in System Design*, 19:291–314, October 2001. 14, 16

[36] Robert Kurshan. Formal Verification in a Commercial Setting. In *Design Automation Conference (DAC)*, pages 258–262, 1997. 2

[37] Timo Latvala. Efficient model checking of safety properties. In *Proceedings of the 10th international conference on Model checking software*, SPIN'03, pages 74–88, Berlin, Heidelberg, 2003. Springer-Verlag. 16, 117

[38] Orna Lichtenstein and Amir Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '85, pages 97–107, New York, NY, USA, 1985. ACM. 15

[39] Mikko Lipasti, Christopher Wilkerson, and John Shen. Value Locality and Load Value Prediction. In *Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, pages 138–147, 1996. 51

[40] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[41] Panagiotis Manolios. Correctness of Pipelined Machines. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *Lecture Notes in Computer Science*, pages 161–178. Springer, 2000. 53

[42] Panagiotis Manolios. A Complete Compositional Reasoning Framework for the Efficient Verification of Pipelined Machines. In *International Conference on Computer-Aided Design (ICCAD)*, pages 863–870. IEEE Computer Society, 2005. 53

[43] Kenneth McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993. 2

[44] Kenneth McMillan. Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking. In *Computer Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 110–121. Springer, 1998. 53

[45] Robin Milner. An Algebraic Definition of Simulation Between Programs. In *Proceedings of the 2nd international joint conference on Artificial Intelligence*, pages 481–489, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc. 9

[46] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM. 91

[47] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. 39

[48] Silvia Müller and Wolfgang Paul. Making the Original Scoreboard Mechanism Deadlock Free. In *Fourth Israeli Symposium on Theory of Computing and Systems (ISTCS)*, pages 92–99. IEEE Computer Society, 1996.

[49] Silvia Müller and Wolfgang Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[50] Lawrence Paulson. *ML for the Working Programmer*. Cambridge University Press, New York, NY, USA, second edition, 1996. 89

[51] Amir Pnueli. The Temporal Semantics of Concurrent Programs. In *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, pages 1–20, London, UK, 1979. Springer-Verlag. 6, 12

[52] Amir Pnueli. The Temporal Semantics of Concurrent Programs. *Theoretical Computer Science*, 13:45–60, 1981.

[53] Jun Sawada and Warren Hunt. Trace Table Based Approach for Pipelined Microprocessor Verification. In *Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 364–375. Springer, 1997. 52

[54] Hazem Shehata and Mark Aagaard. A General Decomposition Strategy for Verifying Register Renaming. In *Design Automation Conference (DAC)*, pages 234–237. ACM, 2004. 53

[55] John Shen and Mikko Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw Hill, first edition, 2004. 44, 48, 49

[56] Jurij Šilc, Borut Robič, and Theo Ungerer. *Processor Architecture: From Dataflow to Superscalar and Beyond*. Springer, 1999. 46

[57] A. Prasad Sistla. Safety, Liveness and Fairness in Temporal Logic. *Formal Aspects of Computing*, 6:495–511, 1994. 14

[58] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. In *12th Annual Symposium on Switching and Automata Theory*, pages 114–121, October 1971. 16

[59] James Thornton. Parallel Operation in the Control Data 6600. In *Fall Joint Computer Conference*, volume 26, pages 33–40, 1964.

[60] James Thornton. *Design of a Computer: The Control Data 6600*. Scott, Foresman and Co., Glenview, Illinois, 1970.

[61] Robert Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11:25–33, 1967.

[62] Moshe Vardi. Branching vs. Linear Time: Final Showdown. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin / Heidelberg, 2001. 6

[63] Moshe Vardi and Pierre Wolper. Reasoning About Infinite Computations. *Information and Computation*, 115:1–37, November 1994. 15

[64] Miroslav Velev. Automatic Formal Verification of Liveness for Pipelined Processors with Multicycle Functional Units. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 3725 of *Lecture Notes in Computer Science*, pages 97–113. Springer, 2005. 52

[65] Pierre Wolper, Moshe Vardi, and A. Prasad Sistla. Reasoning About Infinite Computation Paths. In *24th Annual Symposium on Foundations of Computer Science*, pages 185–194, November 1983. 15