Automatic Formal Verification of Control Logic in Hardware Designs

by

Zaher S. Andraus

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2009

Doctoral Committee:
Professor Karem A. Sakallah, Chair
Professor John P. Hayes
Professor Romesh Saigal
Associate Professor Todd M. Austin
Assistant Professor Chandrasekhar Boyapati
Professor Randal E. Bryant, Carnegie Mellon University

© Zaher S. Andraus

All Rights Reserved

2009

To my Dad and Late Mom

Table of Contents

Dedication ii

List of Tables v

List of Figures vi

List of Appendices vii

Abstract viii

Chapter 1 Introduction 1

Chapter 2 Background 7
- 2.1 Abstraction-Based Verification 8
  - 2.1.1 Verification based on Datapath Abstraction 8
  - 2.1.2 Verification based on Property-Driven Abstraction 11
- 2.2 SAT-based Verification 12
  - 2.2.1 SAT Reduction 12
  - 2.2.2 Unsatisfiability Proof Extraction 14

Chapter 3 An Approximation-based Framework for Hardware Verification . . 16
- 3.1 Problem Formulation 16
- 3.2 A Scheme for Approximation 18
- 3.3 Verification based on Approximation 21

Chapter 4 Verification Based on Datapath Abstraction and Refinement 26
- 4.1 Verilog Descriptions 27
- 4.2 Term-based Abstraction Framework 28
- 4.3 Abstraction to the EUF and CLU logics 33
  - 4.3.1 Abstraction to EUF 33
  - 4.3.2 Abstraction to CLU 40
- 4.4 Property Specification and Validity Checking 41
- 4.5 Counterexample-Guided Refinement 43
- 4.6 Soundness 44

4.6.1 A Sound Abstract-then-Unroll Process with UCLID/Vapor 45
4.6.2 A Sound Unroll-then-Abstract Process in Reveal 54

Chapter 5 Enhanced Abstraction and Refinement 56
5.1 Lemma-based Refinement 56
5.1.1 Generalized Lemmas 56
5.1.2 Explanation of Infeasibility 60
5.1.3 DP-CEGAR 61

Chapter 6 Reveal: An Implementation of DP-CEGAR 63
6.1 Reveal's Software Design 63
6.1.1 Hardware Relations 64
6.1.2 The Formula Generator 65
6.1.3 The Solver 66
6.1.4 The MUS Extractor 67
6.1.5 The CEGAR Core 67
6.2 A Designer's Perspective 68
6.2.1 Reveal's Input 68
6.2.2 Reveal's Output and Counterexample Traces 69

Chapter 7 Experimental Studies 71
7.1 Sorter Case Study 73
7.2 Instruction Cache RAM Case Study 75
7.3 Out-of-Order Memory Updates Case Study 78
7.4 DLX Case Study 79
7.5 RISC16F84 Case Study 83
7.6 X86 Case Study 85
7.7 MIPS Case Study 88
7.8 Experimental Observations 96
7.8.1 Datapath and Memory Abstraction 96
7.8.2 Refinement Trade-Offs 97
7.8.3 Overall Scalability 98
7.8.4 Discovering Design and Specification Bugs 99

Chapter 8 Conclusions and Future Work 100

Appendices 103

Bibliography 137

List of Tables

Table

4.1 Symbolic Unfolding of Memory using Lambda Expressions 31

7.1 Benchmark Statistics 72
7.2 Verification Condition Stats 77
7.3 Verification Results for DLX 81
7.4 Verification Results for RISC16F84 84
7.5 Verification Results for X86 87
7.6 MIPS Bubble-to-Bubble Bug Reference 92
7.7 MIPS Spec-to-Impl Student Verification Results 94
7.8 Verification Results for DLX, RISC16F84, and X86 97
7.9 Verification Condition Nodes and Bits Stats 98

List of Figures

Figure

1.1 Progression in Microprocessors: Design versus Verification 2

3.1 Sound and Complete Approximations 20
3.2 Iterative Approximation/Correction 22
3.3 An "Unabstractable" Equivalence Cricuit 24

4.1 A Dual Port Memory in Verilog 31
4.2 Applying EUF Abstraction to Common Design Components 35
4.3 Datapath Abstraction in a Pipelined Impl. of a MIPS-like Microprocessor . 36
4.4 An Example Design and Property 38
4.5 A Circuit Representation of the Design and Property 39
4.6 Unrolling and Abstraction 44
4.7 Verilog Example Illustrating the Usage of Bit Fields 51
4.8 The Abstraction of the Bit Fields of 'word' 52
4.9 UCLID Abstraction from Verilog 53

5.1 Data Structures for the Counterexample Generalization Algorithm 58
5.2 Counterexample Generalization Algorithm 59
5.3 The DP-CEGAR Algorithm 61

6.1 A Snapshot from a Counterexample Trace Arising during MIPS Verification 70

7.1 Sorter Test Case 73
7.2 Runtime Graphs for Sorter 75
7.3 ICRAM and X86 Testcases 76
7.4 Runtime Graphs for OMU 79
7.5 MIPS Specification and a Naïvely Pipelined Implementation 89
7.6 MIPS Full Pipeline and Equivalence Circuitry 90

List of Appendices

Appendix

A VERSA Verilog 104
A.1 Verilog 95 104
A.2 Synthesizable Subset 105
A.3 Clocking 105
A.4 Explicit Description 105
A.5 Structural Description 106
A.6 Abstraction-Oriented Description 106
A.7 Memories 107

B Configuration Directives 108
B.1 Algorithmic Behavior 108
B.2 Input Specifications 109
B.2.1 Design Modeling 109
B.2.2 Design Information 110
B.3 Output Specifications 111
B.3.1 Back-end 111
B.3.2 Debugging 111

C MIPS Bubble-to-Bubble Counterexample Trace 113

D MIPS Spec-to-PipeCounterexample Trace 124

Abstract

Our work addresses the challenge of scaling pre-silicon functional verification of hardware designs such as microprocessors and microcontrollers. These designs employ wide datapaths with arithmetic, logical, and memory units, and complex control logic that coordinates their functionality. This overall complexity results in an enormous state space with vast room for design errors, and prevents designers from being able to comprehensively reason about the correctness of digital systems deployed in numerous devices, whose failure causes serious losses, monetary and otherwise.

In particular, control optimizations play a global role in coordinating the functionality and data flow. This makes them extremely error-prone and harder to verify locally. To remedy that, a design implementation can be verified against its full specification model which has a much simpler control logic. Then, a formal proof of equivalence exhaustively checks the state space for potential control bugs, or proves the lack thereof. Contrary to simulation-based approaches, which compromises completeness for speed, formal equivalence is hindered by the exponential state explosion. To overcome that, previous approaches abstract datapath components away in order to eliminate the complexity introduced by them, and to gear the verification towards the control logic. Due to the loose separation between datapath and control in most designs and hardware description languages, naïve abstraction results in compromising the accuracy of the verification, and in generating spurious behavior that does not exist in the original design, or masking real behavior from being represented in the abstract model.

Our work presents a systematic and fully automatic abstraction-based method to over-

come these issues. A sound abstraction to fragments of first-order logic is coupled with refinement mechanisms that adjust the abstract model and prevent false alarms arising due to spurious behavior. Our approach includes novel techniques for analyzing abstract counterexamples, generalizing them to represent families of counterexamples, checking their feasibility on the original design, and analyzing the infeasibility in order to learn facts that augment the abstraction in an iterative process. Automating both the abstraction and refinement steps without compromising scalability gives our approach a clear advantage over systems that require laborious manual reasoning. In turn, the approach can be easily embedded in typical verification flows, where designers apply it on original descriptions used for synthesis and traditional simulation.

Additionally, our approach leverages the advantages of efficient reasoning engines for Boolean and first-order logic, including satisfiability (modulo theories) solvers and algorithms for minimal explanation of constraint infeasibility.

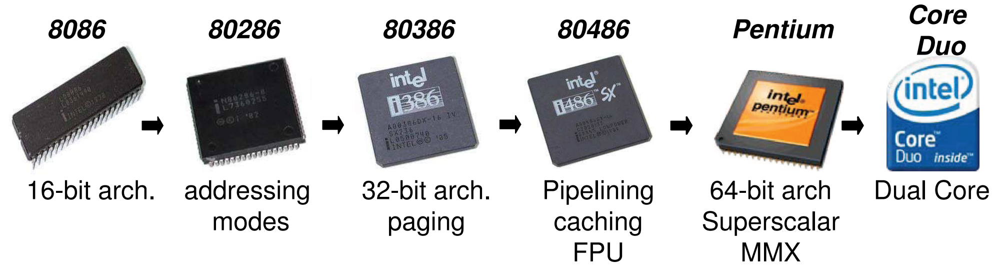
An implementation of the approach allows us to verify microcontrollers, microprocessors, and memory systems whose RTL Verilog descriptions have hundreds to thousands of source lines and variables, in a scalable and efficient manner. The results show promising capability in exposing implementation and specification errors, or, alternatively, proving correctness of both. We believe that this brings formal verification one step closer to hardware designers.

Chapter 1
Introduction

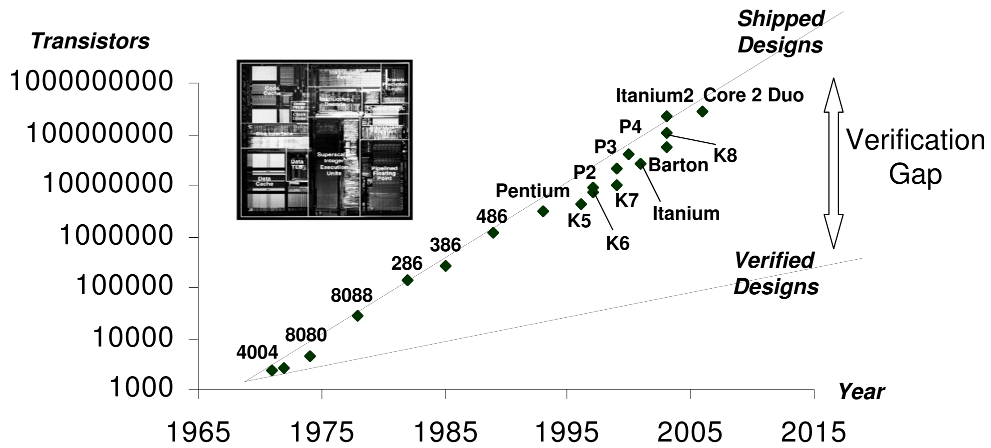
This work addresses the challenge of automating and scaling pre-silicon functional verification of state-of-the-art hardware designs, such as microprocessors and microcontrollers. These designs employ wide *datapaths* with arithmetic, logical, and memory units, and complex *control logic* that coordinates their functionality. The latter typically includes a set of high-level optimizations aimed at increasing a design's throughput, and reducing its area and power consumption. The complexity of both the datapath and control logic results in an enormous state space with vast room for design errors. Furthermore, the progression in the design of hardware systems like microprocessors leads to ever-increasing control logic complexity, and overall chip size, as predicted by Moore's law [45].

In contrast to *simulation*, which typically examines a (relatively) small number of scenarios when testing a design, *formal verification* systematically proves correctness of a design by exhaustively examining the entire design's state space searching for violations to a well-specified behavior. The size of the state space grows exponentially with the size of the design, leading to the so-called *state explosion problem* [19]. Since the control logic and datapath of contemporary designs are also growing exponentially (in both size and complexity), the formal verification 'barrier' grows doubly exponentially, and significantly lags behind the design capability, leading to an exponentially growing *verification gap*. The increase in complexity and size of today's designs, as well as the difficulty of formally verifying these designs, is pictorially expressed in Figure 1.1.

Verification, thus, cannot be made tractable without a divide-and-conquer approach



(a) Architecture features in Intel's microprocessor series



(b) Transistor count[a] of shipped versus verified designs[b]

[a]The statistics are based on various publicly available sources.
[b]The line illustrates the scalability trend. Exact verification measures are yet to be known.

Figure 1.1 Progression in Microprocessors: Design versus Verification

that tailors different verification methodologies to various parts of the design with different structural patterns. To be effective, these methodologies must be applied at suitable levels of abstraction. In particular, descriptions given at the Register-Transfer Level (RTL) accurately capture the functionality of hardware designs by preserving high-level semantic information that is otherwise lost when moving to the gate- or transistor-level representations. It is, therefore, reasonable to assume that the design under verification be given as an RTL model in a suitable Hardware Description Language (HDL) such as Verilog [54].

At this level, a reasonable distinction can be made between the datapath and the control logic, and appropriate verification schemes can be applied to each. Datapath units can usually be isolated and verified separately with methods that exploit their structural regularity.

Once verified, many datapath elements can be reused across various designs and architectures. Control logic, on the other hand, globally "routes" the flow of data in a design, and thus has to be verified at the level of the entire design. Moreover, control circuitry is invariably custom-made for the architecture and design at hand, precluding the use of previous verification results. *In this work we focus on the verification of control logic*.

Current verification efforts have tackled control logic verification by generating new mathematical models, typically based on abstraction, that correspond to the RTL description of the design, and utilizing theorem provers [17][27][34][49] to reason about them. Although these models simplify the datapath, they are roughly as complex as the original RTL model. Consequently, hundreds of man-hours are required to manually regenerate the verification model from the RTL model. Moreover, a cumbersome process is required to keep both models consistent, and to prevent subtle bugs from being introduced in the abstract model or masked from the RTL model.

Theorem provers use a number of mathematical approaches to certify that a design complies with its desired functionality, and typically incorporate a number of theories, ranging from zero-, to first-, to higher-order logics, to incrementally prove correctness. In addition to the drawbacks of verifying an abstract model separately from the RTL description, theorem provers are not fully automatic; although equipped with a set of engines on their back-end, the user is required in many cases to guide the prover by applying specific engines in the various phases of the proof. In the best case, manual reasoning significantly impedes the verification task for complex designs, and in the average case it makes it completely infeasible.

The approach we advocate in this thesis significantly differs from previous efforts in a number of ways, including the emphasis on full automation, as well as the use of contemporary reasoning engines that have been (and will continue to be) progressing tremendously. We believe that the combination of both elements is a key enabler to the deployment of formal verification in design flows.

Our approach achieves scalability of control logic verification through three main theoretical contributions presented in this thesis. First, we formalize an approximation-based framework for hardware verification, that unifies formal, semi-formal, and simulation-based approaches. In this framework, various types of approximations can be characterized based on their relation to the original design and the property being checked.

Second, we formalize datapath abstraction of a subset of synthesizable RTL Verilog, which enables formal verification of hardware designs to scale in space and time. In particular,

- The abstraction process maps a Verilog description with finite-size bit vector variables and operations to an infinite-size term-level model. The resulting abstract model can be expressed using a number of quantifier-free First-Order Logic fragments, per the designer's choice.
- The abstraction function is a conjunction of 'local abstractions' to datapath components. This enables fine-grained control over the granularity of the abstraction, and allows tailoring different abstractions to various parts of the design.
- The abstraction process is geared towards equivalence checking, where abstracting away datapath components allows meaningful reasoning and property checking of the control logic.
- The abstraction function is sound with respect to bounded model checking of safety properties.
- The abstraction function is not theoretically complete, due to datapath/control interactions typical in almost all designs described at the word level. However, in equivalence-based verification, idiosyncrasies arising from these interactions can be localized and eliminated.

Third, we present a method that detects and eliminates 'false alarms' (also called infeasible counterexamples) arising in coarse incomplete abstractions. The refinement in-

crementally ‘fixes’ the abstraction to make it complete, or determines the existence of a corresponding real bug in the design. More specifically,

- The refinement includes novel techniques for relating abstract counterexamples, i.e. property violations in the abstraction, with the original design. This is done through a satisfiability formulation, on which SAT reasoning engines are applied.
- We introduce the use of minimally unsatisfiable subsets to distill concise explanations for infeasibility. We make a novel use of the resulting explanations during refinement.

These techniques were folded into a practical turn-key verification system for RTL Verilog, such that

- The datapath of the design is abstracted, while the control logic is left concrete. This tailors the approach towards detecting control bugs.
- The approach is tailored to verifying RTL descriptions. This is important since designers are usually reluctant to rewrite their design for the purpose of formal verification. Verifying the design source code is intuitively a safe and effective method for the detection of bugs.
- Reasoning is done using a sophisticated Satisfiability Modulo Theories (SMT) solver to prove the validity and satisfiability of the various formulas arising during verification. Formalizing the various reasoning challenges using SMT semantics allows leveraging the tremendous state-of-the-art advances in this domain, and creating additional challenges for future research.
- The approach can be fully automated; a preliminary implementation of the verification system allows the designer to specify the design description and a brief set of directives, and the system formally proves correctness or disproves it with a counterexample.

The Reveal system, a C++ implementation of the presented verification approach,

allowed us to experiment with RTL designs written in Verilog. The ability to fully automate the abstraction and refinement steps gives our approach in particular, and formal verification in general, an edge over other techniques. The goal is to be able to verify microcontrollers, microprocessors, and memory systems whose RTL Verilog descriptions have thousands of source lines and variables, in a scalable and efficient manner.

The rest of the thesis is organized as follows. Chapter 2 surveys relevant work in formal hardware verification, and presents relevant definitions and notations. Chapter 3 presents a general framework for verification based on approximation, followed by an abstraction/refinement method in Chapter 4 specifically tailored to the verification of control logic in RTL designs. Chapter 5 introduces a number of techniques that augment the refinement process and significantly boost its convergence. Reveal's structure and crucial implementation details are presented in Chapter 6, followed by a demonstration of Reveal's ability to verify seven design benchmarks in Chapter 7. Finally, Chapter 8 concludes our work with a summary of our contribution and proposes research directions that may potentially scale the approach further.

Chapter 2

Background

In this chapter we survey relevant work in the area of formal verification of control logic in hardware designs, and particularly in microprocessors in particular. Over two decades of research in these areas has made it possible to address the challenges of verifying state-of-the-art designs, with feasible and practical solutions.

Early efforts in this area attempted to answer two main questions:

- What is the mathematical model that represents the (specification and implementation of the) design to allow scalable verification?
- How should the verification criterion be formulated? i.e., how should the correct behavior of the design be defined?

Due to the tremendously large state space of a microprocessor's behavior, most verification methods have incorporated one or more forms of abstraction. Therefore, the first question involves choosing the right level of abstraction to reduce complexity, while preserving enough information to allow meaningful verification. The choice of the abstraction method has implications on the size of the resulting model, and the needed "mechanical tools", mainly mathematical reasoning methods, to complete the verification task. The second question touches on the, somewhat philosophical, definition of "what it means to be a correct design". Our work focuses on the first question[1].

[1] Examples of methods that focus on the second question include those of Aagaard *et al.* [1], Bose and Fisher [9], Burch, Dill, and Windley [15][16][63], Bryant *et al.*[14], Hosabettu *et al.* [32], Manolois *et al.* [39][40][42], Sawada and Hunt [50][51], Srivas and Bickform [53], and Velev *et al.* [57][59] .

2.1 Abstraction-Based Verification

In general, methods for the verification of control logic in microprocessors and microcontrollers can be categorized into two types. The first type utilizes the structural pattern in a design's description, to differentiate the datapath from the control logic, and in turn to apply aggressive abstraction of the datapath. We will refer to this type as *Datapath Abstraction*. The second type, which we refer to as *Property-driven Abstraction*, models the design as, roughly speaking, one finite state machine, and incorporates the specific property being checked in the abstraction process. We will describe the relevant work in both categories, noting that datapath abstraction has been, generally speaking, more prevalent in the literature of microprocessor verification.

2.1.1 Verification based on Datapath Abstraction

In this type of verification, the abstraction is applied to the datapath, while in most cases control logic is left at the concrete Boolean level. Initial methods were based on theorem provers; later methods evolved by increasing automation and scalability. The following briefly describes these efforts in chronological order.

Manual Abstraction and Verification

The verification of full-fledged microprocessor control logic dates back to 1990; Srivas and Bickform [53] used the Clio Theorem Prover to verify Mini Cayuga, a simplified version of the Cayuga RISC microprocessor developed at Cornell University. Both the 3-stage pipeline and an abstract unpipelined specification were modeled in Clio at an abstract "instructions execution" level. Part of the verification task was facilitated by Clio, which performed automatic symbolic execution, expression normalization, generalization and induction. Large portions of the task, however, needed manual intervention for modeling and verifying the design. The verification was done with manual case splitting, formulation of

criteria, and mechanical proofs; the total effort was estimated to be one man-year. Datapath elements were manually abstracted into black boxes with well-specified behaviors. A similar approach was taken by Cyrlik [24] using PVS [49]. Datapath units were represented using integers, rationals, higher-order logic, and programming language structures such as arrays, lists, and segments. Widley and Coe [62] used the HOL [17][27] theorem prover to model the datapath with higher-order logic.

Automatic Verification

The work by Burch and Dill [15] in 1994 represented a breakthrough in terms of automation and scalability. The authors suggested modeling the datapath with fully black-boxed units, called uninterpreted functions[2], and presented a solver for a logic they dubbed EUF, or equality with uninterpreted functions. These UFs "similarly" abstract datapath elements in the pipelined implementation and the unpipelined specification of the microprocessor. This allows the designer to isolate and independently verify datapath elements, then "plug them back in" as UFs, enabling an automatic reasoning engine to discover bugs in the control logic, which is left unabstracted. This work spawned a number of subsequent efforts including the work of Sawada and Hunt [50], Lahiri *et al.* [37], and Velev *et al.* [11][55][56][58][57][3]. These approaches were used to verify microprocessors with architectural features as complex as out-of-order execution, hardware interrupts, multi-cycle datapath units, and branch predictors. However, they all suffered from limited automation in formulating the verification criterion, as well as the abstraction of the datapath. Sawada and Hunt's work, for example, used the ACL2 theorem prover [34] to verify the control logic of the DLX microprocessor using EUF, requiring three months of manual effort for the abstraction and verification.

Along the lines of datapath abstraction, CMU's UCLID language [12] was designed to

[2]Corella [22] was the first to suggest that UFs can be used to abstract away details of the datapath.
[3]The main difference among these methods was in the formulation of the verification criterion.

facilitate the modeling of microprocessors. Specifically, a UCLID model allows the user to express the interactions of the datapath and control logic as a word-level state machine. In addition, the datapath can be modeled abstractly in the CLU logic, which enhances EUF with limited form of counting and efficient memory modeling. Based on the UCLID language, the authors proposed a number of methods to automatically decide CLU formulas within the UCLID tool. In addition to the UCLID group at CMU, Manolios and Srinivasan [40] were the first to use UCLID to verify the control logic of an XScale-like processor. Lahiri *et al.* also used UCLID to verify an out-of-order microprocessor [32].

Automatic Abstraction

Despite their merits, UCLID models have to be "derived" from micro-architecture descriptions that are written in an HDL such as Verilog [54]. In [4] we made the pragmatic assumption that designers would be unwilling to manually generate a UCLID model for verification purposes, since this necessitates laborious analysis of the RTL model, as well as incremental updates to the UCLID model whenever the RTL model is updated. This not only leads to longer verification iterations, but also to the possibility of injecting additional bugs due to human errors, while hiding real bugs in the original RTL model. Instead, the work in [4] showed that Verilog models can be automatically "abstracted" to produce UCLID models that are suitable for verification. Similarly, Hojati and Brayton [31] translate RTL Verilog to an ICS (Integer Combinational Sequential Concurrency) model, which describes hardware systems at a high level of abstraction using integers, interpreted and uninterpreted functions.

It is worth mentioning that none of the previously described methods incorporates automatic refinement techniques on the abstracted elements. When the abstraction is too coarse to reason about the correctness criterion, false negatives arise and have to be eliminated. Thus, verification based on these methods proceeds iteratively by manually refining the abstraction until the correctness condition can be established or a genuine design bug that

violates it is found. The first automatic refinement technique for the EUF and CLU logics was introduced in [2][3][5], and forms the basis of the approach described in this thesis.

2.1.2 Verification based on Property-Driven Abstraction

While the automation of datapath abstraction is relatively recent, property-driven abstraction has been thoroughly studied in the context of model checking by Clarke *et al.* [21] and Cousot and Cousot [23] for over two decades. Moreover, automatic refinement has been studied as well, wherein false negatives are automatically checked and eliminated, making the abstraction both sound and complete. This process is often referred to as Counterexample-Guided Abstraction Refinement (CEGAR for short), and it has been shown to be an effective paradigm in a variety of hardware, and even software, verification scenarios. Originally pioneered by Kurshan [36], it has since been adopted by several researchers as a powerful means for coping with verification complexity. Clarke *et al.* [20], Jain *et al.* [33] and Das *et al.* [25] have successfully demonstrated the automation of abstraction and refinement in the context of model checking for safety properties of hardware and software systems. In particular, these approaches create a smaller abstract transition system from the underlying concrete transition system and iteratively refine it by eliminating spurious counterexamples due to the incompleteness of the initial abstraction. These methods have been successfully used in practical systems, such as the Microsoft SLAM project [7][6] and the Synopsys RFN Tool [60], and research in this domain is still actively on-going.

The abstraction in this category is of two types. The first type, referred to as localization reduction [18][20][29][44][61], abstracts the transition system by hiding state variables, i.e. replacing the driving logic of some registers with “don’t cares”. The second type, referred to as predicate abstraction [28], abstracts the transition system by projecting it onto a finite set of predicates that are relevant to the property being checked. Localization reduction is mainly suitable for hardware verification since the reasoning is done at the level of bits.

Predicate abstraction, on the other hand, reasons about "words of data" and was originally adopted to verify software. The use of predicate abstraction in hardware verification has been done recently by Jain *et al.* [33], and was tailored for designs expressed in Verilog.

2.2 SAT-based Verification

2.2.1 SAT Reduction

Since the introduction of model checking and theorem proving to the arena of formal verification, researchers have used reductions to propositional and first-order logic, and used reasoning engines as back-end tools. In particular, the use of Binary Decision Diagrams (BDDs) [13] and SAT solvers [43][46] made these reductions a quite feasible option for formal verification methods, both as a theoretical framework and for designing practical and efficient tools. For example, several synthesis and verification methods reduce the original problem to solving a SAT instance, and apply off-the-shelf SAT solvers on it. Almost all recent verification approaches rely on satisfiability solvers as their back-ends. In this section we formulate the satisfiability problem, introduce commonly-used notations, and survey available solutions.

Solving Boolean SAT

Boolean (more accurately, 2-valued) satisfiability solvers have made tremendous practical progress over the past decade, despite the fact that they tackle an NP-complete problem. Given a set of constraints over Boolean variables, the SAT problem seeks a satisfying assignment to the variables that is consistent with all the constraints, or outputs UNSAT if no such assignment exists. The most commonly-used formulation refers to a conjunction of constraints, each (called a clause) represented as a disjunction of literals, where a literal is a 2-valued variable or its logical negation. This format is referred to as Conjunctive Normal

Form (CNF). For example, the CNF formula $(x1') \wedge (x1 \vee x2) \wedge (x1 \vee x2' \vee x3)$ conjoins a 1-literal, 2-literal, and 3-literal clauses[4]

Contemporary SAT solvers such as MiniSAT [26] are able to handle CNF instances with tens of thousands of variables and millions of constraints. Thus, it has been common practice to reduce verification problems to CNF instances on which an off-the-shelf SAT solver can be invoked.

Solving Segments of First-Order Logic

Theorem provers incorporate different techniques for solving First-Order Logic (FOL) formulas by combining 'theory solvers'. The earliest methods are attributed to Nelson and Oppen [47] and Shostak [52]. The latest generation of these solvers target the decidable subset of FOL, and is referred to as Satisfiability Modulo Theory (SMT) solvers. These solvers integrate the theory solvers within a backtrack propositional solver, thus being able to take advantage of the high-level semantics of the non-propositional constraints (e.g., EUF constraints) while at the same time benefiting from the powerful reasoning capabilities of modern propositional SAT solvers.

A different approach to solving quantifier-free FOL formulas relies on reduction to SAT. The original formula is converted to an equi-satisfiable propositional formula using a suitable encoding, which is then checked for satisfiability by a Boolean SAT solver. A satisfying assignment is then mapped back to the original formula. This family of solvers was mainly developed for deciding the validity of EUF and CLU [11] formulas. The main conceptual difference between these encoding-based solvers and SMT solvers is that the former encode all the constraints required by the logic beforehand, while the latter incorporates relevant constraints in an on-demand fashion. The incremental and cooperative framework of SMT algorithms allows them to be extensible to many theories such as equalities, UFs,

[4] The formula can also be described as $(x1')(x1+x2)(x1+x2'+x3)$. This more compact notation will be used throughout the thesis.

linear and non-linear arithmetic, and fixed-size bit-vectors. This gives SMT-based solvers an edge over direct reductions to Boolean SAT.

2.2.2 Unsatisfiability Proof Extraction

Explaining the unsatifiability of SAT instances is an important challenge for several SAT-based applications, including formal verification of hardware. Given a CNF formula, an Unsatisfiable Sub-formula, or US for short, is the conjunction of an unsatisfiable subset of the formula's constraints. A Minimally Unsatisfiable Sub-formula (MUS) is a US such that it becomes satisfiable when any constraint is removed. For example, the CNF formula $(x1)(x1')(x2)(x2')(x1+x2)$ has 3 MUSes, namely $(x1)(x1')$, $(x2)(x2')$, and $(x1')(x2')(x1+x2)$. Any subset that includes one or more of these MUSes is a US. Extracting USes and MUSes from unsatisfiable formulas was first used in [64] to "explain" the unsatisfiability of CNF formulas. Two algorithms, zCore and zMinimal, were designed to find a US and an MUS, respectively, from an UNSAT formula. The AMUSE [48] and CAMUS [38] tools were later introduced and were geared towards performance and finding multiple small MUSes. The latter can also find all MUSes of a given UNSAT formula.

When a SAT algorithm is invoked on a CNF instance that encodes high-level constraints, proving its unsatisfiability indicates the infeasibility of the original constraints. Thus, extracting USes and MUSes of an unsatisfiable formula can be used to "diagnose" the infeasibility of the original problem. In the context of CEGAR, USes and MUSes have been utilized to analyze the infeasibility of abstract counterexamples [2][33] that arise when verifying the abstraction. In these cases, the abstract counterexample represents a scenario that is allowed by the abstraction, but is disallowed when taking into account the constraints of the concrete design. USes can thus pin-point the causes of infeasibility and determine the required refinement of the abstraction, in order to eliminate the infeasible counterexample in the current iteration. Identifying these constraints allows CEGAR-based algorithms to automatically refine the abstraction and resume the verification. Furthermore, using one

or more MUSes during this analysis helps in identifying a larger number of concrete constraints that are crucial for checking the correctness condition, which if left abstracted will cause additional infeasible counterexamples. Predicting infeasible counterexamples, and eliminating them as early as possible in the refinement loop, helps to reduce the number of abstraction/refinement iterations in CEGAR-based methods.

Chapter 3

An Approximation-based Framework for Hardware Verification

In this chapter we explore a verification approach that analyzes the structure of the design, and automatically derives an *approximation* on which state-of-the-art verification algorithms can be applied. This chapter will first formalize the verification task, and will then describe a generic approximation framework and its use as a practical approach to verification. For such an approach to be scalable, the approximation should be significantly easier to verify than the original design, and such verification should yield meaningful conclusions about the correctness of the original design. In this chapter we show how the approach can be adapted in order achieve these objectives; we do so by setting up a generic approximation framework, whose specifics are refined in Chapter 4.

3.1 Problem Formulation

Our framework assumes that the design is given as a reactive transition system, which is described via sequential and combinational hardware components that are connected to the inputs and outputs of the design. Each component is characterized by a so-called *consistency function* that characterizes its functional behavior by relating its outputs and inputs with appropriate constraints. In addition to the design description, the framework requires a sequential bound k, such that the correctness of the design is proven only up to that bound. While requiring a known bound may seem to limit the utility of the approach, empirical ob-

servation suggests that it has application in many situations where such bounds are known a priori or can be easily derived from the particular structure of the design. Examples include verification of pipelined microprocessors, packet routers, and dataflow architectures common in filters, etc.

Given the design's description and the bound k, *unrolling* is used to derive a purely combinational description of the design's transition relation. This process is linear in k and the size of the design. If we let X denote the set of variables in the unrolled description, then the consistency function of each interconnected component i can be described by a constraint $E_i(X)$, and the formula $exact(X) \doteq \bigwedge_{1 \leq i \leq n} E_i(X)$ characterizes the entire behavior of the unrolled design. In most cases, verification is done by comparing an implementation design to a specification "design". In these cases $exact(X)$ includes the unrolled versions of both designs.

The verification problem can then be phrased as the question of establishing the *validity* of the formula $exact(X) \rightarrow prop(X)$, where $prop(X)$ indicates a specified *correctness condition*[1]. An equivalent, but slightly more convenient, form of this formula is $exact(X) \cdot (p = prop(X)) \rightarrow p$, where p is a free variable (not in X) that represents the property being checked. In this form, the sub-formula $(p = prop(X))$ can be viewed as the consistency function of the correctness property in the same way that $exact(X)$ is the consistency function of the concrete design. For simplicity, however, and bearing in mind that it can be considered part of $exact(X)$, we will omit $(p = prop(X))$ from our formulas.

Checking the validity of $exact(X) \rightarrow p$ is typically done by checking the satisfiability of its negation:

$$\varphi(X, p) = exact(X) \cdot \neg p. \tag{3.1}$$

Proving the unsatisfiability of (3.1) establishes that the property holds, while a satisfying solution $(X^*, 0)$ demonstrates the existence of a design or specification bug.

[1] Also know as "verification criterion"

3.2 A Scheme for Approximation

Except for trivial designs, checking the satisfiability of (3.1) directly is generally infeasible. Instead, in the proposed scheme the property is verified on an *approximation* of the exact design, which is a partial representation of the design's functionality. For such an approach to work, the approximation must, by construction, be significantly easier to verify than the original design, both computationally and practically. It also must be related to the original design in such a way that verifying it allows deriving conclusions about the original design. Along these lines, we will introduce a generic notion of soundness and completeness with respect to a property p, which are useful for deriving suitable approximations as we will show later. Throughout these definitions, we will use $M(X)$ to denote a conjunction of constraints over X that models the design, either exactly or approximately. For brevity, we will omit M's explicit dependence on X.

Definition (Relative Soundness and Completeness) if $(M_1 \rightarrow p) \rightarrow (M_2 \rightarrow p)$ is valid (i.e., holds true for all assignments to X), where M_1 and M_2 are two models of the design, then M_1 is called a sound approximation of M_2, and M_2 is called a complete approximation of M_1.

It is easy to show that soundness, as well as completeness, are transitive, reflexive, and anti-symmetric relations, therefore defining *partial orders* over the possible models. Since completeness and soundness are dual, we will unify the two orders and use $\prec$ such that $M_2 \prec_p M_1$ if M_1 (M_2) is a sound (complete) approximation of M_2 (M_1).

If we let E denote the constraints that exactly model the original design (i.e. $E \doteq exact(X)$), then a soundness and completeness notion can be defined for approximations of E as follows.

Definition (Soundness and Completeness) Approximation A is called sound (complete) if $E \prec_p A$ ($A \prec_p E$).

Approximations that are sound, complete, or both, can be very useful, particularly when it is significantly easier to check the validity of $(A \rightarrow p)$ than to check the validity of $(E \rightarrow p)$ and still draw meaningful conclusions about E. Approximation-based methods are, therefore, based on the idea of deriving an approximation, checking the property on it, and drawing conclusions on the original model. For example, if the property holds on a sound approximation, it will definitely hold on the original model, since $(E \prec_p A) \wedge (A \rightarrow p) \rightarrow (E \rightarrow p)$[2]. Conversely, if the property is violated on a complete approximation, it will definitely be violated on the original model.

We can reason about the space of sound *and* complete approximations by simplifying the expression in *Relative Soundness and Completeness* definition:

$$\begin{aligned}(M_1 \rightarrow p) \rightarrow (M_2 \rightarrow p) &\equiv (M_1' + p)' + (M_2' + p) \\ &\equiv M_1 p' + M_2' + p \\ &\equiv M_2' + M_1 + p \\ &\equiv M_2 \rightarrow M_1 + p \\ &\equiv M_2 p' \rightarrow M_1\end{aligned}$$

and replacing M_1 (M_2) with A (E) to represent soundness, and with E (A) to represent completeness:

$$Soundness : E \prec_p A \Rightarrow (A \rightarrow p) \rightarrow (E \rightarrow p) = E \rightarrow A + p = \underline{Ep' \rightarrow A}$$
$$Completeness : A \prec_p E \Rightarrow (E \rightarrow p) \rightarrow (A \rightarrow p) = \underline{A \rightarrow E + p} = Ap' \rightarrow E$$

Therefore, any sound and complete approximation A must satisfy $Ep' \rightarrow A \rightarrow E + p$, which can also be written as $A \in [Ep', E + p]$. In this interval, we have great latitude in choosing the approximation, as illustrated pictorially in Figure 3.1.

[2]This can be shown easily from the *Relative Soundness and Completeness* definition.

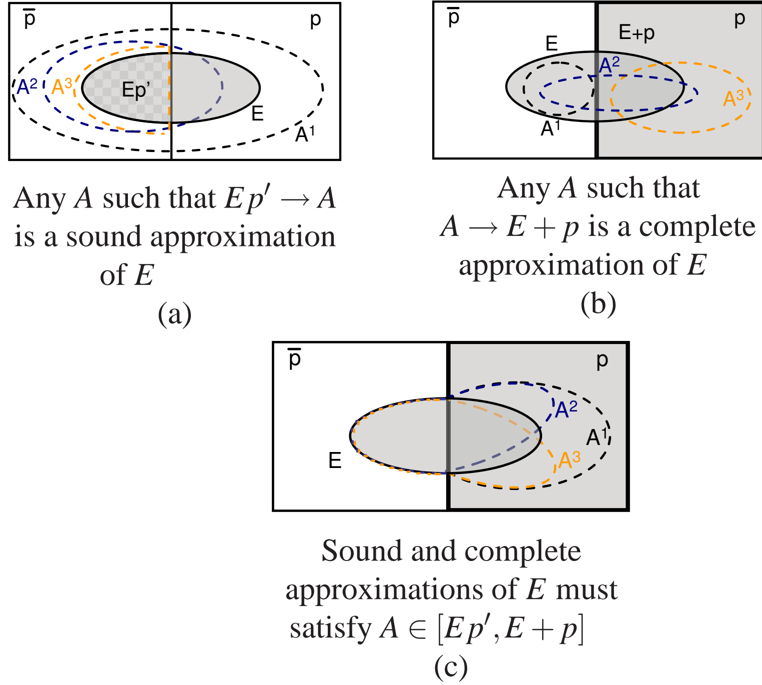


Figure 3.1 Sound and Complete Approximations

Our final set of definitions introduce over- and under-approximations, which are special cases of sound and complete approximations, respectively.

Definition (Relative Over- and Under-Approximation) if $M_1 \rightarrow M_2$ is valid, then M_2 over-approximates M_1, and M_1 under-approximates M_2.

Similarly to sound and complete approximations, over- and under-approximations define partial orders over the possible models, and are represented with the operator $\prec$. Finally, A is called *an over-approximation* if $E \prec A$, and *an under-approximation* if $A \prec E$.

It is important to note that over- and under-approximations can be defined without referencing the property p under consideration. As we will shortly show, this will be very useful in our framework.

3.3 Verification based on Approximation

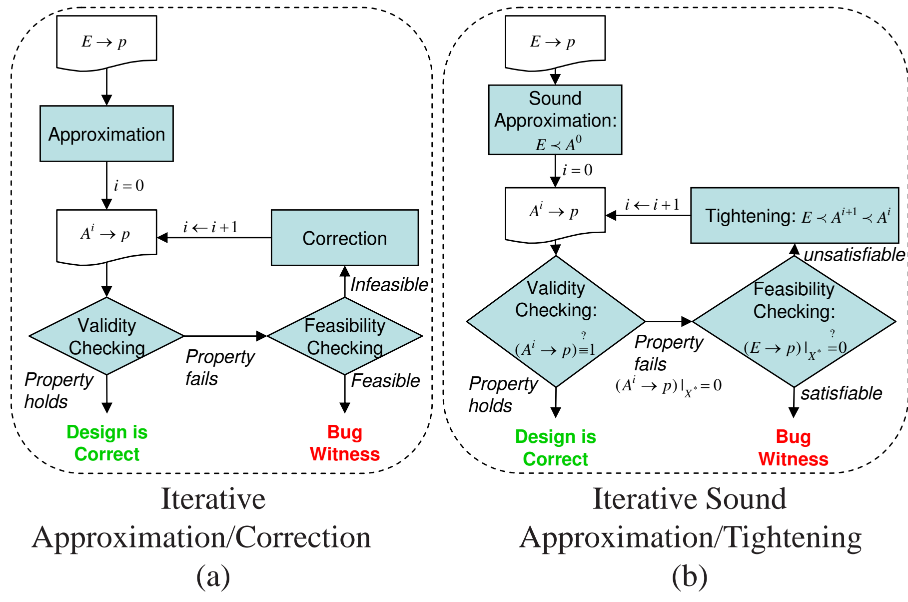
As mentioned earlier, the goal of verification methods that use approximations is to find a sound *and* complete approximation that simplifies the verification task. Theoretically, finding such an approximation is as hard as solving the original problem. Practical algorithms, instead, start with an approximation A^0 of E that is either sound or complete (but not both), and check the property on it. Then, a sequence of more accurate approximations $A^1, A^2, \ldots, A^k$ is iteratively generated until the property can be proven to fail or hold on E.

Figure 3.2 highlights the main steps in this iterative approximation/correction approach. In this figure, diagram (a) illustrates the generic approach, whereas diagram (b) shows a special case based on *sound* approximations. The algorithm starts with a sound approximation A^0 that is not necessarily complete (i.e. $E \prec A^0$). Then, through *incremental tightening*, a sequence of more accurate approximations $A^k \prec A^{k-1} \prec \ldots \prec A^1$ is generated such that

- In iterations $i = 0, \ldots, k-1$, the property is violated on A^i but not on E. The violation witness, also called a counterexample, represents a *false negative* clearly indicating that A^i is not complete. A new approximation A^{i+1} is then derived such that $E \prec A^{i+1} \prec A^i$.
- In the last iteration k, one of two scenarios takes place: the property is violated on both A^k and E; or the property holds on A^k , i.e. $A^k \rightarrow p$. Note that soundness is preserved throughout the process of tightening the approximation. Therefore, if the property holds on the last approximation, then it also holds on the exact model.

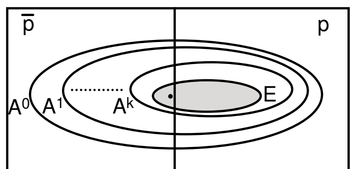
Diagrams (c) and (d) in Figure 3.2 illustrate this verification process in the space of assignments, when the sound approximations used are over-approximations.

In a dual algorithm, an initial complete approximation is verified against the property. If the property is violated on the approximate model, it is definitely violated on the original design; otherwise, no conclusion can be made and the approximate model is *relaxed* to finally obtain a sound approximation for which the property holds.

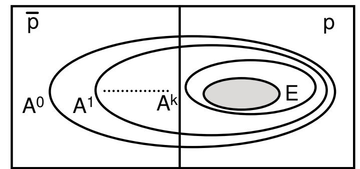


Iterative Approximation/Correction
(a)

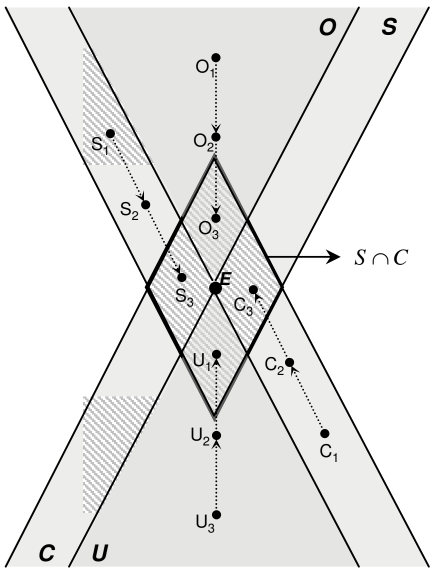
Iterative Sound Approximation/Tightening
(b)



Iterative over-approximation/ tightening leading to a real counterexample
(c)



Iterative over-approximation/ tightening leading to proving the property
(d)



Four flavours of approximation/correction, based on:

(S)ound approximations,
(O)ver-approximations,
(C)omplete approximations, and
(U)nder-approximations.

Note: $E = O \cap U$, $O \subset S$ and $U \subset C$
(e)

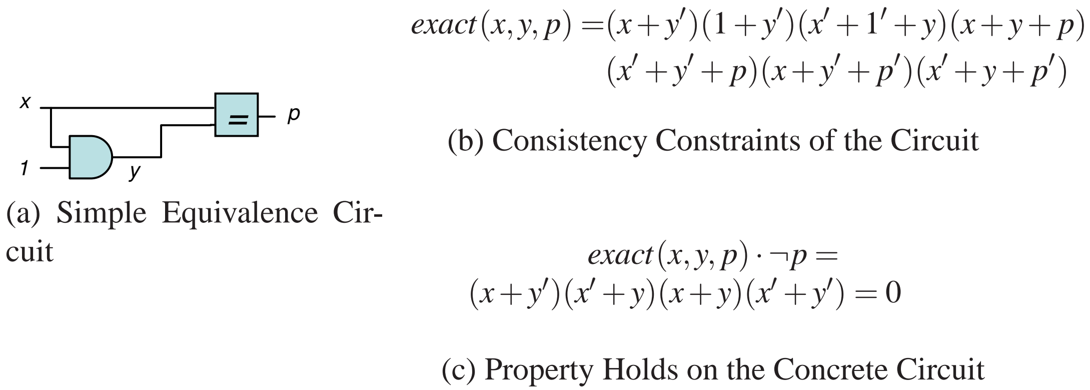
Figure 3.2 Iterative Approximation/Correction

Diagram (e) in Figure 3.2 describes the space of all possible models of the design, such that each point represents a single model, which can be the exact design (E), or a (sound, complete, over-, or under-) approximation thereof. The diagram also demonstrates the relation between these models. For example, every over-approximation is sound, therefore, $O \subset S$. Finally, the diagram shows the various possible approximation/correction processes, particularly tightening (paths $O_1 \rightarrow O_2 \rightarrow O_3$ and $S_1 \rightarrow S_2 \rightarrow S_3$) and relaxing (paths $U_1 \rightarrow U_2 \rightarrow U_3$ and $C_1 \rightarrow C_2 \rightarrow C_3$). As described earlier, the algorithm can either terminate when a sound and complete approximation is obtained (e.g. $S_3 \in S \cap C$), or in an earlier iteration when a real violation is found.

Both approximation-based approaches (i.e. tightening versus relaxing) are used nowadays in many verification contexts, in hardware as well as software. The traditional verification scheme starts with *simulating* the design in order to "hunt" for bugs; simulation is a form of approximation-based verification, since the property is verified given a specific input vector. The behavior of the design for this input vector is a complete approximation of the original design. In this context, a *false positive* refers to the scenario wherein the property is violated by the original design, and the violation is not caught when verifying the approximation (i.e. simulating the design). The presence of false positives is a compromise that designers are willing to make in return for scaling up the verification (i.e. simulation) time. Recent methods (e.g. [60]) guide the simulation based on formal methods, and in turn correct the original approximation to lower the possibility of false positives. This can be considered a form of relaxation.

When reaching a certain level of confidence regarding the correctness of the design, the designer becomes interested in proving the lack of bugs; therefore, the intuitive solution in this case is a "top down" iterative tightening based on sound approximations, such as the algorithm we are presenting.

While an iterative approximation/correction approach is appealing at a conceptual level, its applicability hinges on two main premises. First, the approximation process is



(a) Simple Equivalence Circuit

$$exact(x,y,p) = (x+y')(1+y')(x'+1'+y)(x+y+p)$$
$$(x'+y'+p)(x+y'+p')(x'+y+p')$$

(b) Consistency Constraints of the Circuit

$$exact(x,y,p)\cdot\neg p =$$
$$(x+y')(x'+y)(x+y)(x'+y') = 0$$

(c) Property Holds on the Concrete Circuit

Figure 3.3 An “Unabstractable” Equivalence Cricuit

computationally easy, such that its benefit outweighs the inherit loss of information that requires correction to make it sound and complete. For instance, linear-time over- or under-approximations can be obtained via relaxing or tightening the design’s constraints that are associated with certain components. This can be done independently of the property being checked[3], which gives over- and under-approximations an edge over other types of sound and complete approximations. For example, the full-fledged functionality of an arithmetic unit is replaced with a restricted version that models, say, boundary cases such as an overflow computation or a division-by-zero flag; this creates an under-approximation of the design. Datapath *abstraction*, which concerns us in this thesis, is another example of an easy-to-derive over-approximation that is independent of the property.

The second premise is that there *exists* a sound and complete approximation, that is *significantly* different than the original design, on which the property can be proven to hold or fail. The existence of *any* sound and complete approximation, let alone one that significantly differs from the original design, is not always obvious, nor guaranteed. This is especially true when the approach is confined to a class of approximations, such as over-approximations. For example, consider the circuit in Figure 3.3(a), in which a Boolean variable x is compared to y, another Boolean variable that is equivalent to x through the AND gate. In this case $exact(X) \rightarrow p$, as it can be inferred from the expression in (c).

[3]This is true for a certain class of properties, such as bounded safety.

However, it can be shown by inspection that any over-approximation A obtained by removing one or more constraints from $exact(X)$ is sound but not complete, i.e. $A \rightarrow p$. In this case, and many similar cases, applying iterative over-approximation/tightening is likely to be significantly slower than attempting to establish the validity of the exact formula, since the iterative algorithm will gradually tighten the approximation until it is ultimately identical, or very similar, to it. Therefore, the existence of such a "hoped for" approximation, or the lack thereof, can determine whether applying approximation is beneficial, and can assist the verification engineer in developing an intuition regarding its applicability. In the context of datapath abstraction, our method and experimental results confirm the following conjecture:

Conjecture 1 *An approximation process, wherein similar datapath components in the implementation and specification are abstracted similarly, leads to an over-approximation that is sound and very close to being complete.*

While earlier work (e.g. [12][15][55]) showed the existence of *a* sound *and* complete approximation, and demonstrated how it can be derived manually; our work shows that an *automatically* derived approximation is sound, and in most cases very close to being complete. In turn, it can lead to meaningful verification results without compromising the potential scalability of abstraction.

Chapter 4

Verification Based on Datapath Abstraction and Refinement

In this section we show how an iterative approximation/correction approach can be utilized to verify the complex control logic of hardware designs described at the Register Transfer Level (RTL). Our system performs bounded model checking [8] of safety properties on hardware designs described in Verilog[1]. A typical usage scenario involves providing two Verilog descriptions of the same hardware design, such as a high-level specification and a detailed implementation, and checking them for functional equivalence. Given a Verilog description and a sequential bound k, the system extracts a word-level representation of the design's transition relation and unrolls it k times to create a combinational description of the design, on which the approximation/correction approach can be applied.

In the following subsections we will start with an overview of the Verilog hardware description language [54], and define the verification problem of Verilog descriptions at the RTL. We will follow that with an algorithm based on the approximation scheme described earlier. The approach over-approximates the design by removing datapath-related constraints, and performs verification on the constraints representing the control logic. In this new context, the terms "abstraction" and "relaxation" will be used interchangeably to denote over-approximation, and "refinement" will be used to denote correction or tightening.

[1]Extensions can be applied to other HDLs without compromising the merits of the approach.

4.1 Verilog Descriptions

One of the major differences between RTL and gate-level Verilog is that RTL Verilog descriptions operate at the word-level, i.e. they manipulate words of data, usually referred to as bit-vectors. Datapath elements are usually described using bit-vectors. The control logic, on the other hand, uses single-bit signals to control the computation with the use of multiplexors and logic gates, that are respectively described by conditionals (e.g. *if-then-else* and *switch* statements) and Boolean expressions.

Formally, an RTL Verilog description defines a set of signals R, W, I, and M, respectively denoting the registers, wires, inputs and memories in a flat representation of the design. Each signal in $V = R \cup W \cup I$ can be either single-bit ($V^C \subseteq V$) or multi-bit ($V^D \subseteq V$)[2], and signals in M are multi-dimensional arrays of bits. The interactions of the design components are defined in Verilog via assignments. For example, the Verilog code fragment

```
reg [31:0] r;
always @(posedge clk)
  r <= r1+r2;
```

defines the next state of a 32-bit register $r \in R$ as a function of other signals in the design, i.e. r_1 and r_2.

Let X denote the set of variables in the unrolled description. Each interconnected component with output $x_i \in X$ can be described by the consistency constraint $C(X) = (x_i = f_i(X))$ where $f_i(X)$ defines a Verilog word- or bit-level expression:

[2]The naming convention used here associates single-bit signals with control logic (hence "C") and multi-bit signals with datapath components (hence "D"). The rationale behind this will be clarified in later sections.

$$f_i(X) = \begin{cases} c_i & \text{where } c_i \text{ is a constant} \\ in_i & \text{where } in_i \text{ is an input} \\ op_i(x_{j_1},\ldots,x_{j_n}) & \text{where Verilog operator } op_i \text{ is} \\ & \text{applied to signals } x_{j_1},\ldots,x_{j_n} \in X \end{cases}$$

The consistency constraint $C_i(x_1,x_2,x_3) \doteq (x_3 = x_1 + x_2)$, for example, uses the '+' operator to define a word-level constraint that models a 32-bit adder by equating the signal x_3 with the sum of x_1 and x_2, where x_1, x_2, and x_3 are 32-bit signals. Note that a compound constraint can be used to characterize two or more serially connected components. For example, the constraint $C(x_1,x_2,x_3) \doteq (x_3 = x_1 + x_2 \gg 4)$ uses two word-level operators, namely addition and right-shifting, to compose a constraint that conjoins two simpler ones.

4.2 Term-based Abstraction Framework

A common way to abstract design's elements is to replace them with *terms*, *uninterpreted functions* (UFs), and *uninterpreted predicates* (UPs) [15]. The resulting term-level abstraction maintains the consistency of the removed elements without representing their detailed functionality, and leads to a significant reduction in the size of the design's state space. The *abstraction* step is followed by *property checking* and *refinement*. Property checking determines if the abstracted design satisfies the specified property. Refinement determines if the abstraction was sufficient to establish whether the property holds or fails on the concrete design and, if otherwise, to refine the abstraction accordingly.

As mentioned earlier, term-based abstraction can be viewed as a relaxation of the system of constraints that characterize the concrete design. Specifically, if each concrete consistency constraint $C_i(X)$ is relaxed to a corresponding abstract consistency constraint $A_i(\hat{X})$, where X and $\hat{X}$ denote the concrete design signals and their corresponding abstractions, we can model the abstract design by the formula $abst(\hat{X}) = \bigwedge_{1 \leq i \leq n} A_i(\hat{X})$. Note that

$conc(X) \rightarrow abst(\hat{X})$[3].

Formally, we introduce $\alpha(\cdot)$ to denote the abstraction process. $\alpha(\cdot)$ maps the concrete variables and operators to appropriate abstract counterparts. Specifically, if ξ is a concrete variable or operator, its abstract counterpart is denoted by $\alpha(\xi) = \hat{\xi}$. Applying $\alpha(\cdot)$ to the concrete constraint $C(x_1,\ldots,x_k)$ yields $\alpha(C(x_1,\ldots,x_k)) = \alpha(C)(\alpha(x_1),\ldots,\alpha(x_k)) = \hat{C}(\hat{x}_1,\ldots,\hat{x}_k)$. In general, any expression involving concrete variables and operators can be abstracted by recursively applying $\alpha(\cdot)$ to its sub-expressions. For example, applying $\alpha(\cdot)$ to the constraint $C(R_1,R_2,R_3) \doteq (R_3 = R_1 + R_2 \gg 4)$ yields

$$\begin{aligned}\alpha(R_3 = R_1 + R_2 \gg 4) &= (\alpha(R_3) = \alpha(+)(\alpha(R_1),\alpha(R_2 \gg 4))) \\ &= (\alpha(R_3) = \alpha(+)(\alpha(R_1),\alpha(\gg)(\alpha(R_2),\alpha(4))))\end{aligned}$$

Using the mappings $\alpha(+) = add$, $\alpha(\gg) = shift$, $\alpha(R_i) = \hat{R}_i$, and $\alpha(4) = four$, we have $A(\hat{R}_1,\hat{R}_2,\hat{R}_3) = (\hat{R}_3 = add(R_1, shift(\hat{R}_2, four)))$.

Different types of abstraction can be defined based on an appropriate mapping between the concrete constants, variables, and operators, and their abstract equivalents[4]. Furthermore, this approach can take advantage of the design hierarchy, and apply abstraction to the design at different levels of granularity. For instance, an entire datapath unit, such as the ALU, can be replaced with a single UF or UP. Such heterogeneous abstraction can be automated based on syntactic rules, and can also allow manual, yet fairly intuitive, intervention in the abstraction process. While (manual) hierarchy-based abstraction has been mainly used with theorem proving, our approach focuses on automating the abstraction at the level of the design *signals*, and therefore $\alpha(\cdot)$ is defined for each signal in the design.

In addition to abstracting combinational elements with $\alpha(\cdot)$, tractable verification may require the abstraction of memory arrays. Applying only term-based abstraction to an n-word by m-bit memory yields an n-term abstraction. For memories of typical sizes in

[3] $conc(X)$ corresponds to $exact(X)$ introduced in Chapter 3.

[4] Conceptually, this applies also to other abstraction methods.

current designs, n is on the order of thousands to millions of words. Memory abstraction allows modeling an n-word memory by a formula whose size is proportional to the number of write operations, K, rather than to n. Note that memory abstraction is distinct from term-based abstraction. A useful mnemonic device is to think of term and memory abstraction as being, respectively, "horizontal" and "vertical;" they can be applied separately, as well as jointly.

Our system implements memory abstraction using lambda expressions [12]. In particular, the expression $M'(x) = \lambda_x.ite(x = A, D, M(x))$ describes the next state of a memory array M after a write operation with address A and data D. The operator ite is an if-then-else construct simulating a multiplexer. Replacing memory writes with ite expressions and UF applications is performed during the process of unrolling, such that the final formula is lambda-free.

For example, the Verilog code fragment in Figure 4.1 describes the behavior of a 16-word memory that has two ports, one for reading and one for writing, and Table 4.1 describes the state of all the design signals, including the memory array, in the first four cycles of execution after initialization. Two write operations are performed on the memory; the value 1 is written at location 1, and the value 3 is written at location 3. The read port, which always samples the content of the memory in location 1, reads the value stored originally in the memory in cycles 0 and 1, and reads the value 1 starting at cycle 2, due to the write operation that was performed in the previous cycle.

Given these abstraction mechanisms, the algorithm performs the satisfiability check on the abstraction of formula (3.1) (page 17), i.e.

$$\hat{\varphi}(\hat{X}, p) = \alpha(\varphi(X, p)) = \alpha(conc)(\alpha(X)) \cdot \alpha(\neg p) = abst(\hat{X})\alpha(\neg)\alpha(p). \tag{4.1}$$

Using an appropriate abstraction operator, (4.1) can be considerably simpler than (3.1),

```
reg [31:0] Mem [15:0]; // Memory Array
reg [3:0] Addr; // Address of Write Port
reg [3:0] Read; // Address of Read Port
reg [31:0] Data; // Data for Write and Read Ports
reg en; // Enable signal for Write Port

initial begin
   en = 1'b0;
   Addr = 4'd0;
   Data = 32'd0;
end

always[a] begin

assign Read = Mem[4'd1]; // Reading the content of location 32'd1.
```

[a]In most cases, memory writes are synchronized on a clock edge. In this example, however, we omitted any clock specifications to simplify the exposition.

Figure 4.1 A Dual Port Memory in Verilog

Table 4.1 Symbolic Unfolding of Memory using Lambda Expressions

C	Mem[a]	en	A	D	Read [b]
0	$Mem(x) = \lambda_x.M(x)$	0	0	0	$M(1)$ [c]
1	$Mem(x)$ $= \lambda_x.ite(0 \wedge (x = 0), 0, M(x))$ $= \lambda_x.M(x)$	1	1	1	$M(1)$
2	$Mem(x)$ $= \lambda_x.ite(1 \wedge (x = 1), 1, M(x))$ $= \lambda_x.ite(x = 1, 1, M(x))$	0	2	2	$ite(1 = 1, 1, M(1)) = 1$
3	$Mem(x) = \lambda_x.ite(0 \wedge (x = 2),$ $2, ite(x = 1, 1, M(1)))$ $= ite(x = 1, 1, M(1))$	1	3	3	$ite(1 = 1, 1, M(1)) = 1$
4	$Mem(x) = \lambda_x.ite(1 \wedge (x = 3),$ $2, ite(x = 1, 1, M(1)))$ $= ite(x = 3, 3,$ $ite(x = 1, 1, M(x)))$				$ite(1 = 3, 3,$ $ite(1 = 1, 1, M(1))))$ $= 1$

[a]The transition function for Mem in this case is: $Mem'(x) = \lambda_x.ite(en \wedge (x = Addr), Data, Mem(x))$.

[b]The value of the Read port is the instantiation of $Mem(x)$ with $x = 1$.

[c]Since the memory is not initialized in Verilog, the UF M is used to represent the initial memory state.

facilitating its quick solution by a suitable satisfiability checker. Next, we will define the soundness criterion for this abstraction scheme, and then describe two families of term-based abstraction.

To reason about soundness, note that $\alpha(\cdot)$ maps Verilog equality to the interpreted equality predicate between terms. Thus, $\hat{\varphi}(\hat{X}, p)$ has to adhere to two basic rules; equality transitivity, and functional consistency.

Definition (Equality Transitivity) Equality Transitivity w.r.t. any three terms t_1, t_2, and t_3, is defined by the relation $(t_1 = t_2) \wedge (t_2 = t_3) \rightarrow (t_1 = t_3)$.

Definition (Functional Consistency) Functional Consistency w.r.t any two sets of terms $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ and a UF or UP f of arity n, is defined by the relation $[(x_1 = y_1) \wedge \ldots \wedge (x_n = y_n)] \rightarrow [f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)]$.

To relate $\hat{\varphi}(\cdot)$ with $\varphi(\cdot)$, we have to show that the set of constraints arising in $\hat{\varphi}(\cdot)$ are implied by the concrete formula $\varphi(\cdot)$. To do so, we examine the constraints affecting $\varphi(\cdot)$ based on its structure and semantics.

Consider for example the concrete constraint $C(x_1, x_2, x_3) \doteq (x_3 = x_1 + x_2)$ introduced earlier. It embeds three different constraints implicitly; a "fan-in constraint" defining the relation $C(x_1, x_2, x_3) = (x_3 = f(x_1, x_2))$ for some function f; a "domain constraint" defining the possible values of the variables, in this case $0 \leq x_1, x_2, x_3 < 2^{32}$; and a "semantics constraint" defining the exact interpretation of f, which is 32-bit addition in this case. Fan-in, domain, and semantics constraints in $\varphi(\cdot)$ are denoted by $\mathbb{F}$, $\mathbb{D}$, and $\mathbb{S}$, respectively; and their counterparts in $\hat{\varphi}(\cdot)$ are $\hat{\mathbb{F}}$, $\hat{\mathbb{D}}$, and $\hat{\mathbb{S}}$. The relation between these sets is described as follows:

- The abstraction preserves the connectivity of the circuit, i.e. $\hat{\mathbb{F}} = \mathbb{F}$, which can be easily shown to hold in our case from the definition of $\alpha(\cdot)$.
- A necessary condition for $\alpha(\cdot)$ to perform over-approximation is that it is a 1-1 function w.r.t. variables. Consider, for example, $conc = [p = (x = y)]$, and an (erroneous)

abstraction $\alpha_{err}(\cdot)$ such that $\alpha_{err}(p) = p$ and $\alpha_{err}(x) = \alpha_{err}(y) = \hat{z}$. In this case $\alpha_{err}(conc)\alpha_{err}(X) = [p = (\hat{z} = \hat{z})]$, and we can show that $[p = (\hat{z} = \hat{z}) \rightarrow p] \rightarrow [p = (x = y) \rightarrow p]$ does not hold, i.e. $[\alpha_{err}(conc)\alpha_{err}(X) \rightarrow p] \rightarrow [conc(X) \rightarrow p]$ is not true. Therefore, we require in our framework that $\alpha(x) = \hat{x}$ for each variable $x \in X$, where $\hat{x}$ is a "relaxed" version of x, and in turn we have $\mathbb{D} \rightarrow \hat{\mathbb{D}}$. Relaxation, as we will shortly see, can be done by removing the bound constraints altogether.

With these restrictions, $\hat{\varphi}(\cdot)$ over-approximates $\varphi(\cdot)$ if $\mathbb{S} \rightarrow \hat{\mathbb{S}}$. In the next subsections we will define two types of abstraction, and show that the above implication, and in turn soundness, holds true for each type.

4.3 Abstraction to the EUF and CLU logics

4.3.1 Abstraction to EUF

To perform abstraction to the logic of Equality with Uninterpreted Functions [15], dubbed EUF, the set of design signals X is divided into (single-bit) control signals and (multi-bit) data signals, denoted by X^C and X^D respectively. X^C and X^D in the unrolled design represent their counterparts V^C and V^D in the original design description. Generally speaking, and as hinted by the notation, datapath calculations are performed with signals in X^D, whereas control logic is defined with signals in X^C. Classifying a signal as a datapath or a control signal based on its bit width is a syntactic heuristic [4]. However, misclassification of a control signal as a datapath signal or vice versa does not compromise the correctness of the approach. Specifically, a control signal that is abstracted as part of the datapath might yield a spurious counterexample and cause an increase in the number of refinement iterations. The less probable scenario of misclassifying a datapath signal as a control signal causes the abstract model to be unnecessarily detailed and possibly makes the property checking step intractable. Our experimental results show that the overall algorithm is robust

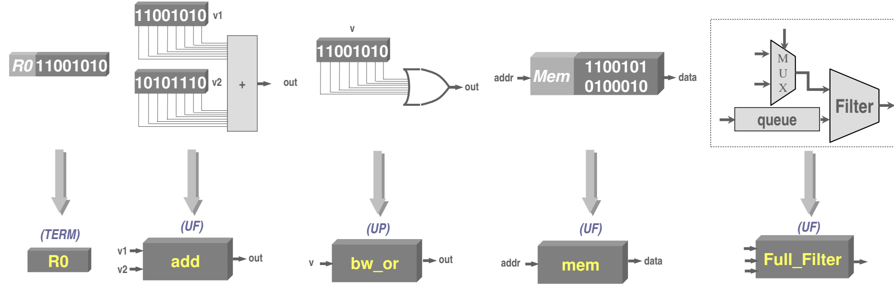
and quite scalable despite control/data intermixtures that may lead to the these scenarios.

By the same token, we will use O^C and O^D to respectively denote the control and data operators. A control operator is one that performs logical operations, i.e. conjunction, disjunction, and negation, on single-bit signals. All other operators are considered data operators. Note that an operator is the occurrence of a symbol in a specific constraint, rather than the mere syntactic token representing it. This is important since Verilog, like other HDLs, defines the semantics of each operation based on its context [54]. For example, the constraint $C_i(x_1,x_2,x_3) \doteq (x_3 = x_1 + x_2)$, introduced in Section 4.1, uses a 32-bit operator to perform addition; the symbol '+' might have different semantics elsewhere. As we will see later, the abstraction process uses 'context' information to determine the abstract counterpart of each Verilog operator.

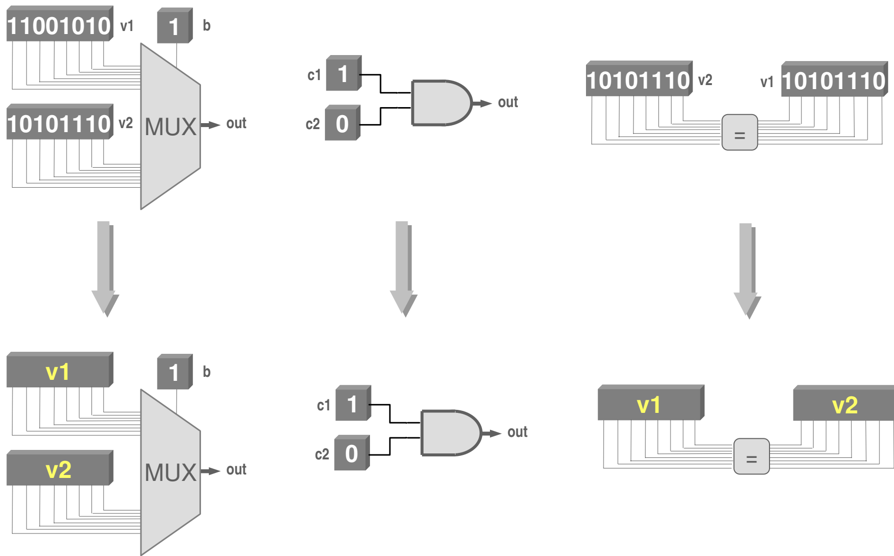
In order to be geared towards control logic verification, *datapath abstraction* removes the detailed functionality of the datapath elements, such as adders, shifters, etc. The interactions among the control signals, however, are preserved making it possible to perform meaningful verification of safety properties on the design's control logic. Along these lines, consider the class of abstractions (based on over-approximation via $\alpha(\cdot)$) that *leave the control logic unabstracted*; i.e., $\alpha(\cdot)$ is the identity function when applied to X^C or O^C. For instance, $\alpha(\neg c) = \alpha(\neg)\alpha(c) = \neg c$. Leaving the control logic in its concrete state preserves enough precision that allows discovering bugs in the control logic.

In this case, $\hat{\varphi}(\cdot)$ is a formula in the quantifier-free first order logic (FOL) defined by the following rules:

1. **Terms**: (a) A non-propositional variable is a term. (b) If f is an n-argument function symbol $(n \geq 1)$ and $t_1,\ldots,t_n$ are terms, then $f(t_1,\ldots,t_n)$ is a term.
2. **Atoms**: (a) A propositional variable (taking values from 0,1) is an atom. (b) If P is an n-argument predicate symbol $(n \geq 1)$ and $t_1,\ldots,t_n$ are terms, then $P(t_1,\ldots,t_n)$ is an atom.
3. **Formulas**: (a) An atom is a formula. (b) If φ and ψ are formulas, then so are $\neg\varphi$,



(a) Datapath abstraction in EUF



(b) Control modeling in EUF

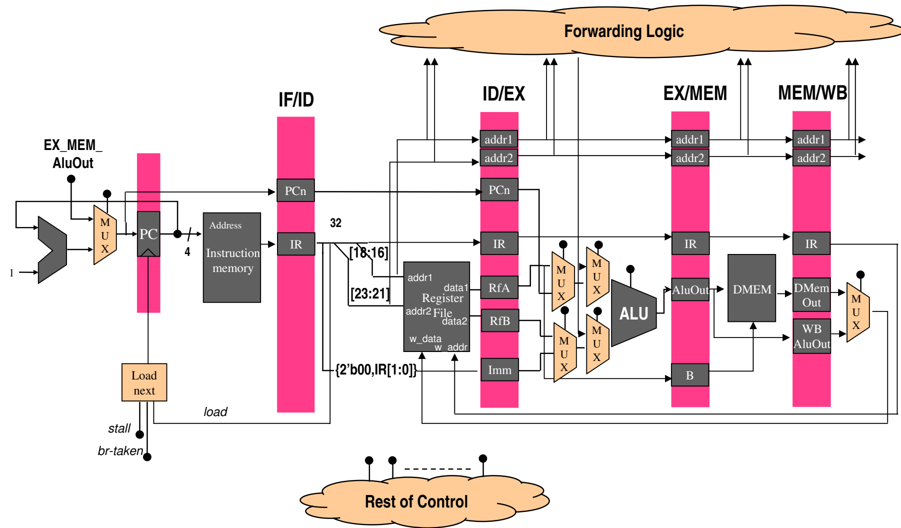
Figure 4.2 Applying EUF Abstraction to Common Design Components

$\varphi \wedge \psi$, and $\varphi \vee \psi$.

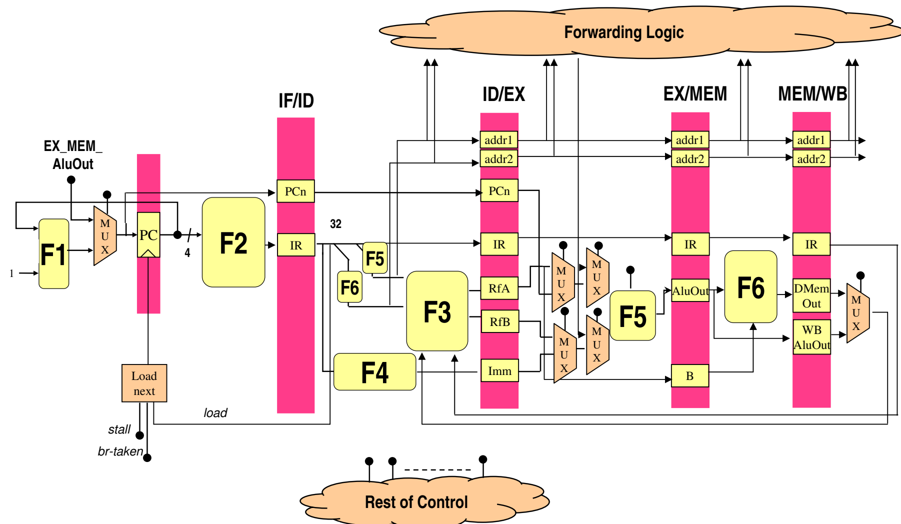
EUF [15] also introduces the if-then-else construct $ite(a, t_i, t_j)$ as an abbreviation for the term which is equal to t_i if the atom a is 1, and is equal to t_j otherwise.

Definition (EUF Abstraction) $\alpha^E(\cdot)$ performs abstraction to the EUF logic by leaving the control logic unabstracted (i.e., modeled via Boolean and *ite* constructs).

Figure 4.2 describes EUF abstraction and its effect on datapath and control logic components, while Figure 4.3 demonstrates datapath abstraction to EUF when applied to a 5-stage MIPS-like microprocessor pipeline.



(a) A 5-stage pipeline with datapath components (gray) and control logic (orange)



(b) Datapath components are abstracted to stubs colored with yellow

Figure 4.3 Datapath Abstraction in a Pipelined Impl. of a MIPS-like Microprocessor

Note that this datapath abstraction mechanism does not determine the way terms are modeled. In particular, since the only interpreted operators acting on terms are equalities, we have freedom in how to model the terms, including leaving them uninterpreted. The choice of the specific abstraction has implications on validity checking, as well as refinement. The SMT solver YICES [65], for instance, treats uninterpreted terms as integers and so does our algorithm. Formally, the abstract term of a datapath signal $d \in X^D$ will be denoted by the integer $\hat{d}$, i.e. $\alpha^E(d) = \hat{d} \in \mathbb{N}$. In the rest of the thesis, we will use $\hat{X}$ and $abst(\hat{X})$ to respectively denote $\alpha^E(X)$ and $\alpha^E(conc)(\alpha^E(X))$. The abstraction of any expression $e \in EXP$ is performed by recursive application of $\alpha^E(\cdot)$ as described earlier.

To illustrate EUF abstraction, consider the Verilog "design" in Figure 4.4. The verification objective is to prove that signal p is always true, indicating that the design satisfies the condition $(a = 0) \rightarrow (d = f)$. The formula representing the concrete constraints of this design can be derived by inspection, and is given in Figure 4.4(b).

Using the semantics of bit-vector operations, such as extraction, concatenation, and shifting, along with the standard Boolean connectives, this formula can be translated in a straightforward fashion to propositional CNF so that it can be checked for satisfiability by standard SAT solvers. In fact, for this simple example it is quite easy for a modern SAT solver to prove that $conc \wedge \neg p$ is unsatisfiable which is the same as saying that $conc \rightarrow p$ is valid.

Our objective, however, is to establish this result using abstraction and refinement. A possible abstraction of this design is given in Figure 4.4(c), where detailed bit-vector operations have been replaced by UP and UF symbols. For example, EX1 is a UP that corresponds to extracting the most significant bit of a, and SR2 is a UF that corresponds to a right shift of b by two bits. Terms in this abstract formula, i.e. variables that correspond to bit-vectors in the concrete formula, are now considered to be unbounded integers. They can be compared for equality to enforce functional consistency but are otherwise uninterpreted having lost their concrete semantics. On the other hand, variables in the abstract formula

```
1.  module design();
2.      wire [3:0] a, b;
3.      wire m = a[3]; // msb
4.      wire l = a[0]; // lsb
5.      wire c = m? a >> 1 : a;
6.      wire d = l? b >> 2 : c;
7.      wire e = m? a : a >> 1;
8.      wire f = l? {2b00, b[3:2]} : e;
9.  endmodule;
10. module property();
11.     wire p = !(a == 0) || (d == f);
12. endmodule;
```

(a) Verilog description

$$
\begin{aligned}
&conc(a,b,c,d,e,f,l,m,p) = \\
&\quad (m = a[3]) \wedge \\
&\quad (l = a[0]) \wedge \\
&\quad (m \wedge (c = a >> 1) \vee \neg m \wedge (c = a)) \wedge \\
&\quad (l \wedge (d = b >> 2) \vee \neg l \wedge (d = c)) \wedge \\
&\quad (m \wedge (e = a) \vee \neg m \wedge (e = a >> 1)) \wedge \\
&\quad (l \wedge (f = \{2'b00, b[3:2]\}) \vee \neg l \wedge (f = e)) \wedge \\
&\quad (p = \neg(a = 0) \vee (d = f))
\end{aligned}
$$

(b) Concrete constraints

$$conc \underset{\gamma^E}{\overset{\alpha^E}{\rightleftharpoons}} abst$$

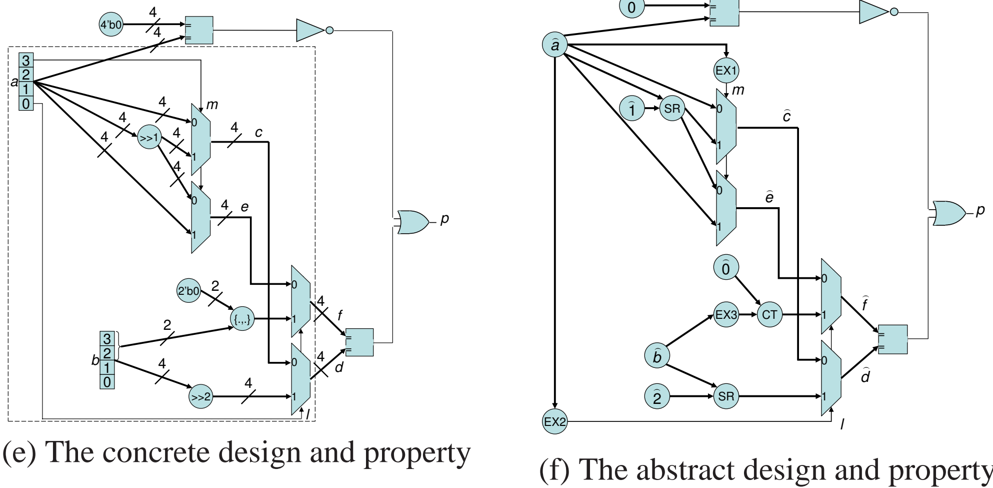
constants	
0	$\hat{0}$
1	$\hat{1}$
2	$\hat{2}$
variables	
a[3:0]	$\hat{a}$
b[3:0]	$\hat{b}$
c[3:0]	$\hat{c}$
d[3:0]	$\hat{d}$
e[3:0]	$\hat{e}$
f[3:0]	$\hat{f}$
l	l
m	m
p	p
operators	
x[3]	$EX1(\hat{x})$
x[0]	$EX2(\hat{x})$
x>>y	$SR(\hat{x},\hat{y})$
x[3:2]	$EX3(\hat{x})$
x,y	$CT(\hat{x},\hat{y})$
!x	$\neg x$

(c) The mapping for α^E

$$
\begin{aligned}
&abst(\hat{a},\hat{b},\hat{c},\hat{d},\hat{e},\hat{f},l,m,\hat{s},\hat{t},\hat{u},p,z\hat{ero},o\hat{ne},t\hat{wo}) = \\
&\quad (m = EX1(\hat{a})) \wedge \\
&\quad (l = EX2(\hat{a})) \wedge \\
&\quad (\hat{s} = SR1(\hat{a}, o\hat{ne})) \wedge \\
&\quad (\hat{t} = CT1(z\hat{ero}, EX3(\hat{b}))) \wedge \\
&\quad (\hat{u} = SR1(\hat{b}, t\hat{wo}) \wedge \\
&\quad (\hat{c} = ite(m, \hat{s}, \hat{a})) \wedge \\
&\quad (\hat{d} = ite(l, \hat{u}, \hat{c})) \wedge \\
&\quad (\hat{e} = ite(m, \hat{a}, \hat{s})) \wedge \\
&\quad (\hat{f} = ite(l, \hat{t}, \hat{e})) \wedge \\
&\quad (p = \neg(\hat{a} = z\hat{ero}) \vee (\hat{d} = \hat{f})).
\end{aligned}
$$

(d) Abstract constraints

Figure 4.4 An Example Design and Property



(e) The concrete design and property

(f) The abstract design and property

Figure 4.5 A Circuit Representation of the Design and Property

that correspond to single bits in the concrete formula (such as m and l) retain their Boolean semantics and can be combined with the standard Boolean connectives.

As mentioned in Section 4.2, a necessary condition for the soundness of $\alpha^E(\cdot)$ is that it is a 1-1 function w.r.t. variables. In fact, $\alpha^E(\cdot)$ has to also be 1-1 w.r.t. operators as well. Consider, for example, the case where $\alpha^E(\cdot)$ maps both $\{x, 3'b000\}$ and $\{x, 2'b00\}$ to $concat(\hat{x}, zero)$. This abstraction is not sound, since the operator $\{\}$ has different semantics in either case; this leads to a scenario wherein $concat(\hat{x}, zero) = concat(\hat{x}, zero)$ is valid, while $\{x, 3'b000\} = \{x, 2'b00\}$ is not. The two expressions should, therefore, be mapped to two distinct UFs under $\alpha^E(\cdot)$. The following lemma articulates that, in the general case, the over-approximation criterion mentioned above is both necessary and sufficient.

Lemma 1 *If $\alpha^E(\cdot)$ is a 1-1 function, i.e. it maps distinct concrete variables to distinct abstract variables, and it maps distinct datapath operators to distinct UFs and UPs, then it is an over-approximation.*

Proof Sketch Let x^* be a full assignment such that $\varphi(x^*) = 1$ and $\hat{\varphi}(\hat{x}^*) = 0$, where x^* and $\hat{x}^*$ assign similar values to corresponding variables in $\varphi(\cdot)$ and $\hat{\varphi}(\cdot)$. $\varphi(x^*) = 1$ implies that $\mathbb{F}(x^*) = \mathbb{D}(x^*) = \mathbb{S}(x^*) = 1$; and since $\mathbb{D} \rightarrow \hat{\mathbb{D}}$ and $\mathbb{F} = \hat{\mathbb{F}}$, we have $\hat{\mathbb{F}}(\hat{x}^*) = \hat{\mathbb{D}}(\hat{x}^*) = 1$, and in turn $\hat{\mathbb{S}}(\hat{x}^*) = 0$. Let the constraint $\hat{s} \in \hat{\mathbb{S}}$ be violated, i.e. $\hat{s}(\hat{x}^*) = 0$. $\hat{s}$ cannot be

an equality transivity constraint, since this type of constraints has to always hold for full assignments. Therefore, $\hat{s}$ is a functional consistency constraint. Since $\alpha^E(\cdot)$ is 1-1 w.r.t. to UFs/UPs and variables, $\gamma(\hat{s})$ involves a single Verilog operator applied on two inputs of equal value under x^*, and producing dis-equal values under x^*; therefore, $\gamma(\hat{s})(x^*) = 0$. Functional consistency has to hold for all operators, i.e. $\mathbb{S} \rightarrow \gamma(\hat{s})$, and since $\gamma(\hat{s})(x^*) = 0$ we have $\mathbb{S}(x^*) = 0$ which contradicts the earlier assumption on $\mathbb{S}(x^*)$. $\square$

In our algorithm, a 1-1 $\alpha^E(\cdot)$ function is enforced with the use of a naming convention for UFs and UPs [4]. In particular, since operator semantics in Verilog are defined by its *operation* as well as the *size of its arguments*, the name of a UF or UP is a concatenation of the operator type and argument sizes. For example, a 32-bit addition is abstracted to the UF called 'add_32_32'.

Finally, it is worth mentioning that since $\alpha^E(\cdot)$ is 1-1, its inverse $\gamma^E(\cdot)$ is well-defined (see Figure 4.4(b)). $\gamma^E(\cdot)$ remaps terms back to their corresponding multi-bit variables, and remaps uninterpreted functions to their corresponding bit-level counterparts. The use of $\gamma^E(\cdot)$ will be evident in the refinement back-end of our algorithm.

4.3.2 Abstraction to CLU

CLU [12] is a quantifier-free first-order logic that extends EUF with separation constraints and lambda expressions. Separation constraints allow the use of limited counting arithmetic useful in modeling certain hardware constructs such as memory pointers. Lambda expressions allow aggressive, albeit consistent, abstraction of memories. Note that we "borrow" Lambda expressions to model memory arrays even when using the EUF logic; thus, the main difference between EUF and CLU in our case is counting. We will use $\alpha^C(\cdot)$ to differentiate CLU abstraction from EUF abstraction ($\alpha^E(\cdot)$).

The use of counting in CLU is done using an interpreted operator $succ^c$ that allows adding an integer constant c to an abstract variable $\hat{x}$. Note that such use would not have

been feasible if abstract variables are represented with non-integer constructs, such as bit-vectors.

In hardware design, there are two frequent occurrences of addition and subtraction of constants. The first occurrence, which is rather implicit, is in the use of any stand-alone constant c; in essence, c is equivalence to $succ^c(\hat{0})$. Constants are used frequently in decoders, such as in IR[3:0]=4'b0101 ($succ^5(\hat{0})$); or in the control logic in counters, such as in cnt==3'd4 ($succ^4(\hat{0})$). The second use of constant addition is in the incrementing of counters, such as in cnt<=cnt+4'd1 ($succ^1(\hat{cnt})$;

In order to remain sound, the abstraction of constant addition with the interpreted *succ* operator in CLU has to guarantee that $\alpha^C(\cdot)$ is a 1-1 function. This is always true in the case of constants; $\alpha^C(c)$ can always be modeled with $succ^c(\hat{0})$ regardless of the size of the bit-vector representation of c in Verilog. The latter is true since any two constants of the same value, but of different bit-width, are still equal according to Verilog semantics[5]

This no longer holds for counting. If $\alpha^C(\cdot)$ is oblivious to the size of x, then it will always abstract $x + c$ with $succ^c(\hat{x})$, although *overflow* occurs differently depending on the bit-width of x and the value c. In general, it is possible to assume that counters do not overflow as done in [4]. In particular, one can rewrite certain counters to remove implicit overflow. For example, the counter cnt<=cnt+1 for a 2-bit variable cnt can be replaced by cnt<=ite(cnt==2'd3,2'd0,cnt+1), and in turn eliminate any possible overflow. In practice, it is quite feasible to require designers to adhere to a coding style that avoids implicit overflow with constant addition and subtraction.

4.4 Property Specification and Validity Checking

Early EUF solvers (e.g. [12][55]) convert $\hat{\varphi}(\cdot)$ to an equi-satisfiable propositional formula using a suitable encoding. On the other hand, Satisfiability Modulo Theories (SMT)

[5]In practice, we use the interpreted addition operator only with small constants. Constants that are greater than a pre-defined threshold are abstracted similarly to variables, in order not to overload the abstract solver.

solvers, such as YICES, operate on these formulas directly by integrating specialized "theory" solvers within a backtrack propositional solver. SMT solvers are, thus, able to take advantage of the high-level semantics of the non-propositional constraints (e.g., EUF constraints) while at the same time benefiting from the powerful reasoning capabilities of modern propositional SAT solvers.

Given the (over-approximated) abstract formula $\hat{\varphi}(\cdot)$, the algorithm checks its satisfiability using an SMT solver. If the solver determines that $\hat{\varphi}(\cdot)$ is unsatisfiable, the algorithm halts concluding that the property holds. Otherwise, an abstract counterexample is produced and the refinement phase is invoked.

In this type of abstraction, where terms are integer variables, a satisfying solution to formula (4.1) (page 30) is an assignment of integers and Booleans, respectively, to the terms and atoms in the variable vector $\hat{X}$:

$$\hat{X}^* = viol(\hat{X}) \doteq \bigwedge_{1 \leq i \leq |\hat{X}|} \hat{x}_i = \hat{c}_i,$$

where $\hat{c}_i$ is the constant value assigned to $\hat{x}_i$ (the ith element of $\hat{X}$). As stated, $viol(\hat{X})$ represents a "point" in the space of possible assignments to the variables of formula (4.1), such that it is consistent with the abstract constraints but inconsistent with the correctness property. We indicate this by introducing the satisfiable "violation" formula

$$\nu(\hat{X}, p) \doteq \hat{\varphi}(\hat{X}, p) \cdot viol(\hat{X}) = abst(\hat{X}) \cdot \neg p \cdot viol(\hat{X}), \tag{4.2}$$

that succinctly captures the result of the validity check.

4.5 Counterexample-Guided Refinement

This section describes the most basic refinement type, that is based on refuting spurious counterexamples. In refutation-based refinement, a spurious abstract counterexample is viewed as "undesirable" behavior, and one or more succinct explanations are used to refine the abstraction for the next round of checking. This is similar to clause recording, or learning, in SAT solvers.

To determine if the violation reported by the validity checker is a real violation, we need to evaluate it on the concrete formula. This step, referred to as feasibility checking, can be accomplished by applying $\gamma(\cdot)$[6] to (4.2) yielding:

$$\gamma(\nu(\hat{X},p)) = \gamma(abst)(\gamma(\hat{X})) \cdot \neg p \cdot \gamma(viol)(\gamma(\hat{X})) = conc(X) \cdot \neg p \cdot cviol(X), \tag{4.3}$$

where $cviol(X)$ s the concretization of the abstract violation. Unlike the rest of the formula elements, concretizing constants is not obvious since the variables in the abstract formula are unbounded integers; some assignments will not, therefore, fit within the bound of the originating concrete bit-vector. However, this problem can be avoided altogether by a more suitable representation of the violation, as explained in the next section.

In general, the process of feasibility checking consists of determining the satisfiability of (4.3). If (4.3) is found to be satisfiable, then the violation reported by the validity checker is a real violation indicating a real design (or specification) bug. If (4.3) is found to be unsatisfiable, then the violation is spurious. This triggers abstraction refinement, which strengthens the abstraction by eliminating this violation from it for the next round of validity checking. We will use a superscript to denote the index of the iteration, such that $\hat{\varphi}^0(\hat{X},p)$ denotes formula (4.1). The ith iteration of the abstraction-refinement loop then

[6] The 'E' superscript of α and γ are omitted in this subsection and when obvious from the context.

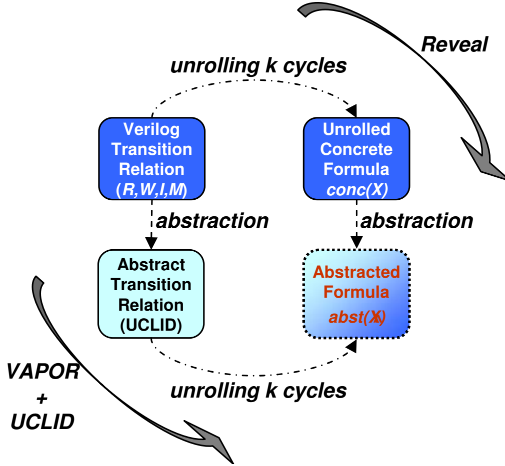


Figure 4.6 Unrolling and Abstraction

consists of the following computations:

1. **Validity Check**: Check the satisfiability of $\hat{\varphi}^{i-1}(\hat{X}, p)$. If unsatisfiable, exit reporting "property holds."
2. **Violation Derivation**: Derive $viol^i(\hat{X})$ from the solution X^* returned in step 1.
3. **Feasibility Check**: Check the satisfiability of $\gamma(\nu(\hat{X}, p)) = \gamma(\hat{\varphi}^{i-1}(\hat{X}, p) \cdot viol^i(\hat{X}))$ If satisfiable, exit reporting "property fails."
4. **Abstraction Refinement**: Compute the new formula $\hat{\varphi}^i(\hat{X}, p) = \hat{\varphi}^{i-1}(\hat{X}, p) \cdot \neg viol^i(\hat{X})$ and go to 1. $\neg viol^i(\hat{X})$ will be called a "lemma" in this framework.

In order for refutation to be practical, steps 2 and 3 have to be computationally easy; and for fast convergence, the violation used for refinement should eliminate as many spurious behaviors as possible. Section 5.1 is dedicated to showing how this is achieved.

4.6 Soundness

This section focuses on the soundness of the abstraction, as well as the interaction between the abstraction and unrolling processes, and the latter's impact on soundness. Figure 4.6 shows two ways of computing an abstract formula $abst(\hat{X})$ using abstraction and unrolling. Starting from a Verilog Transition Relation over the variables R, W, I, and M, unrolling

produces the concrete formula $conc(X)$, which is in turn abstracted to $abst(\hat{X})$ via $\alpha(\cdot)$ as described in previous sections. As illustrated in the figure, the same result can be obtained via producing an abstract transition relation first, followed by unrolling to create $abst(\hat{X})$

In the rest of this section, soundness is explained by:

- Describing a generic approach that performs abstraction to UCLID [12], followed by unrolling and solving.
- Refining the previous approach with the use of Vapor [4] for abstraction. The result is a sound abstract-then-unroll method.
- Describing a sound unroll-then-abstract method that we use in Reveal.

Subsection 4.6.1 is dedicated to explaining the first two methods, while the third is left to Subsection 4.6.2.

4.6.1 A Sound Abstract-then-Unroll Process with UCLID/Vapor

UCLID

The UCLID language allows defining sequential term-based abstract models. This language supports two basic data types, TRUTH and TERM. It also supports two function types: FUNC which maps a list of TERMs to a TERM, and PRED which maps a list of TERMs to TRUTH. These types are combined using operators from the following set:

- Boolean connectives for TRUTH constants and variables.
- Equality ($=$) and ordering ($<$, $>$) relations which operate on TERMs and return TRUTH.
- Interpreted functions *succ* and *pred* which take a TERM and return, respectively, its successor and predecessor. These functions allow modeling counters and represent a limited form of integer arithmetic.
- The ITE (if-then-else) operator which selects between two TERMs based on a

Boolean condition.

- Uninterpreted PRED symbols or Lambda expressions that take TERM arguments and return a TRUTH
- Uninterpreted FUNC symbols or Lambda expressions that take TERM arguments and return a TERM.

A Motivating Example

We will explain the rationale behind a sound Verilog-to-UCLID abstraction with a small Verilog example and a series of improved abstractions. Consider the following Verilog fragment

```
reg [7:0] v;
wire s;
always @(posedge clk)
  if(s)
    v[7:0] <= v[7:0] & 8'h0F;
  else
    v[3:0] <= v[5:2] | 4'h2;
```

As a first-order approximation, the abstraction of such Verilog description to UCLID can be thought of as a syntactic mapping between related variable types in the two languages. For instance, single- and multi-bit signals in Verilog can be mapped, respectively, to TRUTH and TERM variables in UCLID. These mappings, in turn, induce corresponding mappings between Verilog operators and UCLID logical connectives, UFs, and UPs. Such an approach basically assumes that multi-bit signals and the function units that operate on them should be automatically abstracted.

If s is abstracted to TRUTH variable S, and v[7:0], v[3:0] and v[5:2] are respectively

abstracted to the TERMs V_7_0, V_3_0 and V_5_2, a (resulting) intuitive UCLID abstraction, which we call ABST_1, is given below. Note that temporal abstraction of the clock signal 'clk' is modeled with the function 'NEXT', which represents the a single cycle 'advancement' of the transition relation.

```
NEXT[V_7_0]:=ITE(S,AND(V_7_0,const15),V_7_0);
NEXT[V_3_0]:=ITE(!S,OR(V_5_2,const2),V_3_0);
```

Another possible abstraction, which removes more constraints from the UCLID model, and thus is coarser, is:

```
NEXT[V_7_0]:=ITE(S,FREE_V_7_0,V_7_0);
NEXT[V_3_0]:=ITE(!S,FREE_V_3_0,V_3_0);
```

In this abstraction, called ABST_2, arbitrary values are generated using 'free inputs' denoted by the prefix FREE. This bears some similarity to localization reduction [18][20][29][44][61], which abstracts state variables by turning them into free inputs.

Finally, the *coarsest* abstraction will be called ABST_3, and is given by:

```
NEXT[V_7_0]:=FREE_V_7_0;
NEXT[V_3_0]:=FREE_V_3_0;
```

Obviously, ABST_3 is sound, since it is completely unconstrained. It is also easy to see that it is too coarse and does not serve as a meaningful abstraction. A meaningful and sound abstraction is derived similarly to ABST_1 and ABST_2 above, with a counter-intuitive caveat: ABST_1 and ABST_2 are not actually sound for the following reason. When s=0 in the Verilog model, v[3:0] is modified due to the assignment in the 'else' branch, but more

importantly v[7:0] is implicitly modified by virtue of its relation with v[3:0]; on the other hand, V_7_0 remain unchanged in the abstraction when S is false, disallowing a corresponding transition from taking place in the UCLID model. A similar analysis can also be carried out for the case of s=1.

A possible fix is given by $ABST_4$ as follows:

```
NEXT[V_7_0]:=ITE(S,AND(V_7_0,const15),FREE_V_7_0);
NEXT[V_3_0]:=ITE(!S,OR(V_5_2,const2),FREE_V_3_0);
```

This abstraction is similar to $ABST_1$, except that the state of both UCLID variables is 'refreshed' on either sides of the branch. In other words, when the branch is taken, V_7_0 gets assigned a value based on the RHS of the Verilog assignment; *when it is not taken, a fresh arbitrary value is assigned*. A dual abstraction is used for V_3_0.

For this example, soundness is guaranteed by modeling *every* possible transition for each bit field of v. On the other hand, it is possible to constrain the UCLID model further without compromising soundness of the abstraction. In particular, the algorithm in Vapor [4] uses a more 'refined' abstraction, such that the arbitrary symbolic values given by 'FREE' are replaced with UCLID expressions that *relate each bit-vector with its bit fields*. The following section describes the abstraction mechanism in Vapor.

Vapor

As shown in the previous subsection, multi-bit signals typically consist of bit fields that are individually accessed for reading and/or writing. Correct abstraction in such cases must account for the relation among the bit fields and between each bit field and its parent vector. Furthermore, a naïve abstraction may lead to the unintended abstraction of critical control signals that are grouped in Verilog as multi-bit vectors, making the abstract UCLID model too coarse to be usable in verification. Finally, abstraction of certain Verilog operators may

lead to the generation of spurious errors since functional abstraction guarantees consistency under equality but is oblivious to properties such as associativity and commutativity; for example abstracting integer addition with the UF $add(x,y)$ will insure functional consistency but will not treat $add(x,y)$ as identical to $add(y,x)$ as required by commutativity of addition.

The above observations suggest that an abstraction algorithm must not only examine the declared signal types in Verilog but also the way such signals are "used" in the body of the Verilog description. In the rest of this section, we describe how Vapor abstracts various Verilog constructs to corresponding ones in UCLID.

Based on their "bit structure" Verilog variables are classified into three main types. Single-bit variables which are 2-valued and naturally modeled as UCLID TRUTH variables. Multi-bit words which are viewed as unsigned integers and translated into corresponding UCLID TERM variables. Word arrays which typically denote memories or register files and are conveniently represented by UCLID UF variables. Except for the abstraction of bit vectors, these mappings are straightforward. Bit vectors require additional machinery to insure that their abstraction is consistent. Specifically, given a Verilog bit vector X, we must not only create a UCLID TERM to represent X but also create additional TERMs to represent each of its individually-accessed bit fields. Furthermore, we must introduce a set of uninterpreted functions that relate these TERMs to each other. Otherwise, UCLID treats these TERMs as completely independent, potentially leading to the generation of numerous false errors, or to the generation of unsound abstraction.

Without loss of generality, assume that X is a vector of n bits such that $X[n-1]$ is the most significant bit. It is convenient to view X as the interval $[n-1:0]$. Assume further that the set of individually-accessed bit fields of X is denoted by X^F. Thus, X^F is a set of possibly overlapping subintervals of $[n-1:0]$. Finally, let $\pi(X^F)$ denote the coarsest partition of $[n-1:0]$ induced by X^F . For example, if X is $[15:0]$, and $X^F = [15:0],[15:8],[7:0],[10:3]$, then $\pi(X^F) = [15:11],[10:8],[7:3],[2:0]$.

Consistency can now be established by introducing TERMs for each of the bit fields in X^F and $\pi(X^F)$ and a corresponding set of complementary uninterpreted extraction and concatenation functions that relate these TERMs. These functions are designed to insure that whenever a bit field in X^F is changed, appropriate updates are made to all the other bit fields that overlap it. These functions are named according to the naming convension described in Subsection 4.2, in order to insure soundness. In particular, extraction functions are named $extract_m_w(X)$ to indicate the extraction of w bits from bit vector X starting at bit position m[7]. Similarly, concatenation functions are named $concat_w_1_\ldots_w_k(X1,\ldots,X_k)$ to indicate the concatenation of k bit vectors $X_1,\cdots,X_k$ whose bit widths are $w_1,\ldots,w_k$. A similar naming convention is adopted for TERM and TRUTH variables; e.g., the Verilog bit vector $X\,[a:b]$ is declared as the TERM X_a_b.

These notions are illustrated in Figures 4.7, 4.8 and 4.9; which respectively depict a Verilog fragment, bit field relations for 'word', and the corresponding UCLID abstraction. Consider, in particular, how the bit vector $word\,[7:0]$ gets updated. From the Verilog fragment, it is clear that portions of $word\,[7:0]$ are assigned to in both branches of the if statement. Specifically, when mode is equal to 1, the five most significant bits of $word\,[7:0]$ (i.e. $word\,[7:3]$) may change because of the assignment to $word\,[10:3]$. And when mode is equal to 0, $word\,[7:0]$ is assigned the value of w_low. These updates are facilitated by introducing the following UCLID TERMs and associated uninterpreted functions:

- $mode_0_0$, $word_10_3$, and $word_7_0$ to denote the Verilog variable mode, and the individually-accessed bit fields $word\,[10:3]$ and $word\,[7:0]$
- $word_P_2_0$ and $word_P_7_3$ to denote the bit fields of word in the induced partition; $word_P_7_3_n$ is a temporary TERM that denotes the next value of $word_P_7_3$
- the UF $extract_4_5()$ which relates $word_7_3$ to $word_10_3$; $word_7_3$ is derived from $word_10_3$ by extracting 5 bits starting from the fourth most significant bit

[7] Without loss of generality, bit vectors are assumed to be numbered such that bit 0 is in the least significant position.

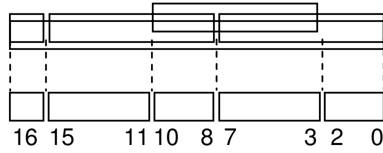
```
// signal declarations
reg [16:0] word;
wire [7:0] w_low,w_high;
wire [16:0] out;
wire parity,clk;

// Verilog fragment with explicit and implicit
// access to bit fields of 'word'
reg mode; always @(posedge clk)
   if (mode == 1'b1)
     word[10:3]<=8'b11001110;
   else
     word<={parity,{w_high,~w_low}};
assign out = word;

// Equivalent Verilog fragment where all implicit
// accesses to bit fields of 'word' are made explicit
always @(posedge clk)
   if (mode == 1'b1)
     word[10:3]<=8'b11001110;
   else begin
     word[16]<=parity;
     word[15:8]<=w_high;
     word[7:0]<=~w_low;
assign out = word;
```

Figure 4.7 Verilog Example Illustrating the Usage of Bit Fields

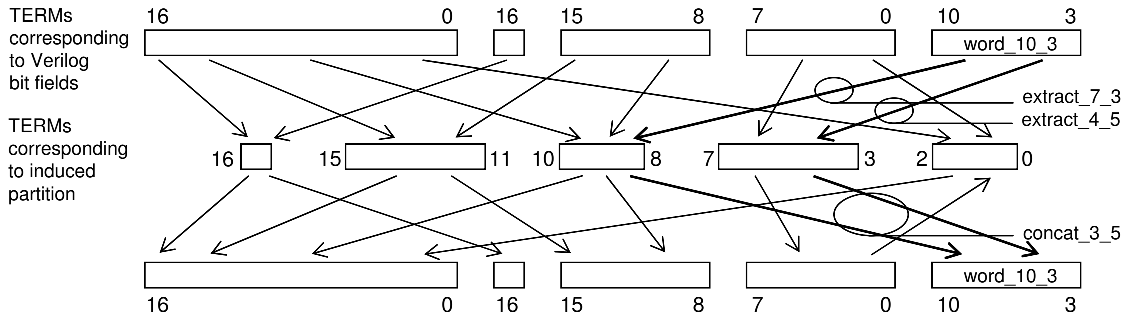
position the UF $concat_5_3()$ which reconstructs $word_7_0$ from $word_P_7_3_n$ and $word_P_2_0$ the UF $bitw_not_8()$ which represents bitwise negation applied on $w_low_7_0$.

The update of $word\,[7:0]$ is now achieved as follows:

1. $word\,[7:0]$ is initialized to some arbitrary symbolic constant (line 27).
2. When mode is equal to 1, $word\,[10:3]$ is assigned an uninterpreted constant value (lines 28 and 29).
3. The next value of $word\,[7:0]$ is set to $bitw_not_8(w_low)$ if mode is equal to 0 (line 35) or, if mode is equal to 1, to the concatenation of the new value of its 5 most significant bits and the old value of its 3 least significant bits (lines 33 and 34).



(a) Partition induced by the bit fields of 'word'



(b) Uninterpreted extraction and concatenation functions needed to insure consistency between 'word' and its bit fields.

Figure 4.8 The Abstraction of the Bit Fields of 'word'

The general scheme described above can be simplified in certain situations and such simplifications can lead to significantly more efficient translations from Verilog to UCLID. For example, if the individually-accessed bit fields of a Verilog bit vector are mutually disjoint, it is not necessary to introduce additional TERMs for the partition blocks. Extraction may also be simplified when applied on constants. These optimizations reduce the size of the propositional formula generated by UCLID since UCLID encodes TERMs using a bit string whose length is a function of the total number of TERMs and UFs applications being processed. Furthermore, we found that such an optimization eliminates many unnecessary false errors by avoiding the need for using extraction UFs.

In the process of obtaining the coarsest refinement over a set of bit vectors, some of the blocks in the resulting partition may end up being single bits. These single-bit fields can be modeled as TERMs and used in extraction and concatenation as described above. This, however, might allow them to get more than 2 different symbolic values. In such cases, we use UPs, instead of UFs, as extraction functions. When the block (TRUTH variable) needs

```
word_10_3 = concat_3_5(word_P_10_8,word_P_7_3)
word_P_10_8 = extract_7_3(word_10_3)
word_P_7_3 = extract_4_5(word_10_3)
```

(a) Uninterpreted functions that act as axioms relating bit field word[10:3] to its corresponding blocks in the partition

```
CONST
  INITS : TERM;
  concat_5_3 : FUNC[2];
  extract_7_3 : FUNC[1];
  extract_4_5 : FUNC[1];
  bitw_not_8 : FUNC[1];
VAR
  mode_0_0 : TRUTH;
  word_16_0 : TERM;
  word_16_16 : TRUTH;
  word_10_3 : TERM;
  word_7_0 : TERM;
  w_low_7_0 : TERM;
  word_P_2_0 : TRUTH;
  word_P_7_3_n : TERM;
  word_P_10_8_n : TERM;
  const53 : TERM;
  ...
DEFINE
  word_P_7_3_n := case
    mode_0_0 : extract_4_5(const53);
    default: ...
  esac;
    ...
ASSIGN
  init[word_7_0] := INITS;
  next[word_10_3] := case
    mode_0_0 : const53;
    default: ...
  esac;
  next[word_7_0] := case
    mode_0_0 : concat_5_3(word_P_7_3_n,word_P_2_0);
    default: bitw_not_8(w_low_7_0);
  esac;
  ...
```

(b) UCLID fragment corresponding to the update of bit field word[7:0]

Figure 4.9 UCLID Abstraction from Verilog

to be concatenated, it has to be “type cast” to TERM, using an appropriate ITE expression.

4.6.2 A Sound Unroll-then-Abstract Process in Reveal

The abstraction and unrolling processes in Vapor examine the coarsest partition of all Verilog variables, maps each bit field in this partition to a UCLID variable, and defines the abstraction based on $\alpha^E(\cdot)$ or $\alpha^C(\cdot)$ as described previously. In this section, we describe an improvement to the abstraction/unrolling processes such that:

- the unrolling and abstraction processes remain sound and close to complete;
- the refinement process automatically strengthens the abstraction in an on-demand fashion, eliminating the need for correlating bit fields upfront, as done in Vapor; and, finally,
- the unrolling-abstraction diagram (Figure 4.6 on page 44) is commutative; i.e., the resulting abstract formula $abst(\hat{X})$ is oblivious to the order of applying unrolling and abstraction.

Consider the Verilog example we introduced in the beginning of this section, involving a register ‘v’ of size 8 bits. The individually-accessed bit fields of v are in this case $v^F = \{[7:0],[5:2],[3:0]\}$, which can be divided into two sets: bit fields accessed in the LHS of any Verilog assignment, denoted by $v^{LF} = \{[7:0],[3:0]\}$, and the rest, denoted by $v^{RF} = \{[5:2]\}$. As illustrated earlier, Vapor examines the coarsest partition entailed by v^F, in this case $\pi(v^F) = \{[7:6],[5:4],[3:2],[1:0]\}$. Reveal, on the other hand, examines the coarsest partition entailed by v^{LF}, i.e. $\pi(v^{LF}) = \{[7:4],[3:0]\}$. The rationale behind this is twofold. First, expressing the change in the ‘state’ of variable v can be done based solely on the LHS accesses. For example, the Verilog code of the running example can be rewritten to the following:

```
reg [7:0] v;
```

```
wire s;
always @(posedge clk)
   if(s) begin
      v[3:0] <= v[3:0] & 4'hF;
      v[7:4] <= v[7:4] & 4'h0;
   end else
      v[3:0] <= v[5:2] | 4'h2;
```

Representing v using $v[3:0]$ and $v[7:4]$, which have no bits in common, allows a straighforward, yet sound, mapping to TERMs v_3_0 and v_7_4 and proceeding with the abstraction. Alternatively, unrolling based on the state of variables $v[3:0]$ and $v[7:3]$ can be done first, followed by the abstraction of the resulting formula. The final abstract formula is similar in either case.

The practicality of the this scheme relies on the fact that Reveal deploys a refinement back-end, allowing any false negatives arising from interacting bit-fields to be resolved automatically. For example, $v[5:2]$ appears only in a RHS expression. This means that (1) it does not need to participate in the state modeling of v, and (2) it can be expressed in the final concrete or abstract formula using v or using the combination of $v[3:0]$ and $v[7:4]$. If false negatives are to arise due to these interactions, the refinement back-end will automatically resolve them.

Chapter 5
Enhanced Abstraction and Refinement

5.1 Lemma-based Refinement

Our experiments show that the implementation details of the abstraction/refinement approach can directly and greatly affect performance. In particular, a number of techniques were found to be crucial for convergence, and essential to the overall performance of the approach. The first group of techniques allow distilling powerful lemmas from abstract counterexamples in a process we refer to as *generalization*. Using these lemmas to refine the abstract counterexample was essential for fast convergence of the refinement loop. The second group of techniques allow generating one or more extremely succinct lemmas in each refinement iteration, and therefore further speeding up the convergence and overall performance significantly.

5.1.1 Generalized Lemmas

The counterexample reported by the validity checker can be viewed as a very specific violation. Checking the feasibility of such a violation is trivial, since it can be done through SAT propagation in equation (4.3). On the other hand, the violation cannot be used to derive a useful refinement since it "encodes" only one particular case, and out-of-bound constants cannot be concretized as described earlier. At the other extreme, the checker can declare that the property is violated, without reporting any information. This corresponds to requiring $viol(\hat{X}) = cviol(X) = 1$, leading to an expensive feasibility check when checking the

satisfiability of (4.3). This, in fact, amounts to doing the verification at the bit level without any abstraction. In this case there is no need for refinement; if (4.3) is satisfiable, a bug is reported; otherwise, the property holds.

In between these two extremes, we have great latitude to choose a suitable representation of the violation, subject to the following objectives:

- It should be efficient to derive;
- It should be efficient to check feasibility on;
- It should provide effective refinement.

We observed that when a violation is detected and checked against the concrete model, only a very small subset of the model's components participate in causing the property to be falsified. A justification process similar to that used in ATPG can identify those constraints (i.e. function boxes, "gates") that participate in the implication chains leading to p being false.

The C-like pseudo-code in figures 5.1 and 5.2 describes the formation of $viol(\hat{X})$ which incorporate four key techniques:

1. Enlarging the footprint of the violation by replacing the concrete assignments to the terms with equalities or inequalities between terms.
2. Creating a very compact representation of $viol(\hat{X})$ based on the *primary inputs* of the design.
3. Excluding the elements of the concrete design that do not fall in the Cone Of Influence (COI) of the violation assignment.
4. Excluding all the control elements (interpreted operators) of the concrete design.

The rationale behind the last technique is that the abstract model automatically accounts for the constraints of the interpreted operators in $\varphi(X)$. Therefore, incorporating these constraints in $viol(\hat{X})$ for feasibility checking overloads the SAT solver with redundant constraints, leading to a potential slow down in the feasibility checking process, as

```
 1. struct {
 2.    string name;
 3.    enum {UF, ite, tvar} type;
 4.    union {
 5.       // list of inputs to UF
 6.       list<term> inputs;
 7.       // inputs to ite
 8.       atom cond;
 9.       term thenterm, elseterm;
10.    }
11.    unsigned value;
12. } term;

13. struct {
14.    string name;
15.    enum {UP, EQ, NOT, OR, AND, pvar} type;
16.    union {
17.       // list of gate inputs
18.       list<term> inputs;
19.       // inputs to EQ
20.       term left, right;
21.    }
22.    bool value;
23. } atom;

24. // either P (P = 1) or !P (P = 0)
25. struct {
26.    // either UP or EQ
27.    atom P;
28.    bool V;
29. } relation;

30. // list of (potential) violations
31. list<relation> viol;
```

Figure 5.1 Data Structures for the Counterexample Generalization Algorithm

```
 1. void EvalFormula(atom f){
 2.    // C-style 'fall-through' switch
 3.    switch (f.type){
 4.      case EQ: {relation r = {EvalTerm(f.left) = EvalTerm(f.right), f.value};
 5.                  viol.insert(r); break;}
 6.      case UP: {relation r = {f.name(EvalTerm(f.inputs)), f.value};
 7.                  viol.insert(r); break;}
 8.      case OR:
 9.      case AND: if (f.value==!controlling(f.type)) EvalFormula(f.inputs);
10.                   else { for (input in f.inputs)
11.                            if (input.value==controlling(f.type))
12.                              {EvalFormula(input); break;}
13.                        }
14.                 break;
15.      case NOT: EvalFormula(f.inputs); break;
16.      case pvar: break; // do nothing
17. } // EvalFormula

18. term EvalTerm(term t){
19.    switch (t.type){
20.      case UF: return f.name(EvalTerm(f.inputs));
21.      case ite: if (t.cond.value==1)
22.                 return EvalTerm(t.thenterm);
23.                 else return EvalTerm(t.elseterm);
24.      case tvar: return t;
25. } // EvalTerm

26. void GeneralizeCE(atom 'abst->prop'){
27.    viol={};
28.    EvalFormula('abst->prop');
29. } // GeneralizeCE
```

Figure 5.2 Counterexample Generalization Algorithm

well as reducing the footprint of the violation.

The algorithm traverses $\hat{\varphi}(\cdot)$, starting from the top node, and recursively invokes the procedures EvalTerm and EvalFormula. Given a term variable t, EvalTerm calculates a symbolic expression representing the value of t when applying X^*, by evaluating the interpreted operators in its sub-tree. EvalFormula is invoked on formulas, including atoms, and it constructs the violation by calculating simplified atoms and their value under X^*. We use the auxiliary function 'controlling', traditionally defined for logic gates as controlling(AND)=0 and controlling(OR)=1.

5.1.2 Explanation of Infeasibility

Given a spurious violation, i.e. $viol(\hat{X})$ such that $conc(X) \cdot \neg p \cdot \gamma(viol(\hat{X}))$ is unsatisfiable, it is possible to further widen the footprint of the learnt lemma by explaining the unsatisfiability of the aforementioned formula via Minimally Unsatisfiable Subsets, or MUSes for short. An MUS is an unsatisfiable subset of the formula that becomes satisfiable if any constraint is removed. The use of MUSes allows the refinement in the current iteration to 'block' violations that might occur in future iterations. Formally, one or more explanations are extracted as follows:

$$
\begin{aligned}
&mus_k(X) = EXPLAIN_k(conc(X) \cdot \neg p \cdot \gamma(viol(\hat{X}))) \\
&abst_mus_k(\hat{X}) = \alpha(mus_k(X)) \\
&expl_k(\hat{X}) = abst_mus_k(\hat{X}) \cap viol(\hat{X})
\end{aligned}
$$

In words, MUS extraction is applied on the UNSAT formula, to explain the infeasibility of the counterexample. We use CAMUS [38] to generate one, multiple, or all MUSes from the formula. The procedure $EXPLAIN_k$ represents the process of extracting the kth MUS, denoted by $mus_k(X)$. Then, the abstraction is used to map the MUS back to the original abstract constraints, and the subset of these constraints originally belonging to $viol(\hat{X})$ is the

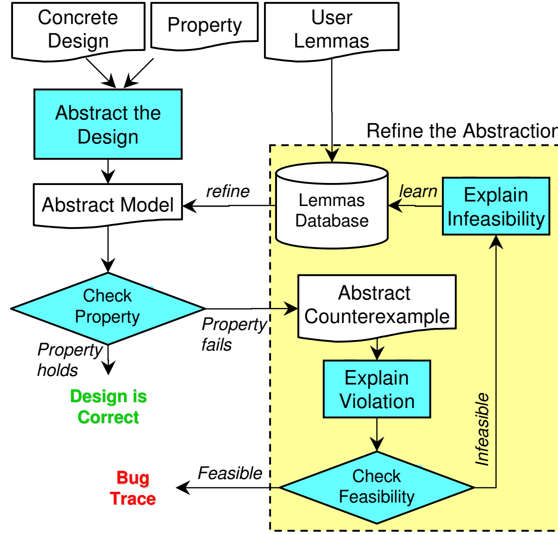


Figure 5.3 The DP-CEGAR Algorithm

final explanation of infeasibility. The refinement, in turn, uses one or more lemmas each represented by $\neg expl_k(X)$. Since $expl_k(X)$ is a subset of $viol(\hat{X})$, the lemma $\neg expl_k(\hat{X})$ is more compact and has wider impact on the abstract model than $\neg viol(\hat{X})$, hence using MUS-based explanation speeds up the refinement convergence as we will see in the experiments. The efficient implementation of MUS extraction in CAMUS and its tight integration with the rest of the refinement algorithm allows this step to remain very fast despite the worst case theoretical complexity of MUS extraction.

5.1.3 DP-CEGAR

Figure 5.3 highlights the overall architecture of our automated verification system, based on Datapath Abstraction and Counterexample-Guided Lemma-based Refinement (DP-CEGAR), as described earlier.

- **Unrolling**. The design is initially unrolled to create a bit-vector formula that represents the design and the property. Unrolling applies a number of optimizations, such as isolating the property's cone of influence, or simplifying memory expressions. Most of these simplifications are orthogonal to the abstraction and are related

to constant propagation in multiplexers.

- **Abstraction and Validity Check**. The abstraction step over-approximates the design's constraints via UFs and UPs, and the resulting formula can be checked using an SMT solver.
- **Refinement**. Abstract counterexamples are checked for feasibility and lead to the generation of lemmas that are stored in a database. The database of lemmas (a) allows the flexibility of aggregating one or more lemmas in each refinement iteration; (b) allows the user to supply lemmas before the verification begins for the design at hand; and (c) allows reusing lemmas across verification sessions invoked on different versions of the same design, or different designs of the same "family of designs".

Chapter 6

Reveal: An Implementation of DP-CEGAR

In this chapter we describe Reveal, a software tool that implements DP-CEGAR, and showcase its usage in identifying (or proving the lack of) design bugs.

6.1 Reveal's Software Design

Reveal is implemented in C++, and consists of the following components:

- The Hardware Relations library described in Section 6.1.1 is a stand-alone package that is used to manipulate word-level expressions. Reveal uses it as its platform for communicating Verilog expressions, as well as abstract lemmas.
- The Formula Generator described in Section 6.1.2 is Reveal's front-end module. It is used to generate $conc(X)$.
- The Solver Module described in Section 6.1.3 is Reveal's back-end module for solving SMT formulas.
- The MUS Extractor described in Section 6.1.4 is another back-end module responsible for extracting infeasibility explanations from SMT formulas.
- The CEGAR Core described in Section 6.1.5 orchestrates the entire process as described in previous chapters.

The following subsections highlight the components' implementation specifics that enable scalability and automation.

6.1.1 Hardware Relations

This module allows Reveal to efficiently store and manipulate word-level expressions throughout the entire flow, as well as to be extensible and applicable to other uses, such as the verification of software. To achieve that, it has to trade off three requirements simultaneously:

- it has to scale in space and time;
- it has to comply with a generic interface that allows interaction with the various components in Reveal, including lemma storage on the desk;
- and finally, it has to allow each different component to store its own metadata, perform its own optimizations on the expressions, and traverse the data structure accordingly.

To achieve that, the implementation was done using a recursive data structure, such that each object represents a word-level expression, which can be a leaf node representing a constant or a variable, or an operation node with an operator and a list of sub-expressions. To scale in space and time, the following techniques are used:

- The library avoids frequent OS calls for memory allocation by internally managing memory allocation and resorting to 'bulk allocation', i.e. asking the OS for larger arrays of free memory at a time. Reveal's performance was found to improve up to 20% when this mechanism overrides C++ native 'new' allocation, especially for large benchmarks with millions of allocated nodes.
- Hashing functions are used to quickly determine whether two nodes represent the same word-level expression, and potentially eliminate resulting duplicates.
- Constant values are automatically propagated through combinational logic and if-then-else expressions, leading to significant simplifications of expressions. Other simplifications include (1) the simplification of trivial multiplexers (e.g. ite(x,a,a); (2) the removal of redundant concatenation/extraction; and (3) the conversion of bit-

wise operations (e.g. Verilog's '$\sim$' operator for negation) on single-bit variables to Boolean logic operations (Verilog's '!' operator in this case).

In order to be extensible, yet support all of Reveal's functionality, the library supports three main functions:

- It allows generic annotation, i.e. storing metadata on each node. Annotation is useful in a number of scenarios. Firstly, it is used to trace a newly created node back to its originating node. For example, assume n_1 is a node representing an expression in the transition function of the design, n_2 is a node representing the value of n_1 in a certain cycle during unrolling, and n_3 is a corresponding simplified expression in the violation. In this case, back-annotation allows n_3 to point back to n_2, which in turn points back to n_1. Secondly, annotation is used to flag nodes during various traversal-based analyses done in Reveal. Thirdly, annotation is used to indicate to the solver or MUS extractor the SMT 'modeling' for each node. The latter will be explained in Subsection 6.1.3.
- It allows recursive traversal of the data structure.
- It allows storage on the desk in the form of a native binary format. This format can be used to store the lemma database, as well as the transition relation of the design after parsing.

6.1.2 The Formula Generator

This module creates equation (3.1) (page 17) by applying the following steps sequentially:

- **Preprocessing.** We use Icarus Verilog [68], an open source simulation and synthesis tool for Verilog, to eliminate compiler directives from the design, such as 'include' and 'define' statements.
- **Flattening and Parsing.** To test the correctness of the design, through simulation or formal verification, the design has to be represented in a so-called *flat view*, such that

all modules and functions are instantiated. Flattening in our case generates a new design that is equivalent to the original, and has no instantiation. The new design is parsed into an in-memory annotated tree using Icarus Verilog [68]. The supported design syntax is given in Appendix A, which is a subset of the overall syntax supported by Icarus Verilog.

- **Calculation of Transition Relation.** The Formula Generator takes the Verilog representation and calculates a transition relation for the design variables based on the method described in Section 4.6.2.
- **Unrolling.** Each Verilog variable in R, W, and M, is assigned a symbolic expression in each cycle based on its transition relation. Rather than using the transition relation as is, the Formula Generator uses the simplifications supplied by the Hardware Relations library to reduce the size of the resulting formula.

6.1.3 The Solver

The Solver module is responsible for determining the satisfiability or validity of FOL formulas. It interfaces with the YICES SMT solver via a C++ API [65]. This module can determine, for example, whether a formula is valid in the EUF or CLU logics, or satisfiable in the bit-vector (BV) logic.

This module makes use of the generic annotation mechanism introduced in Section 6.1.1 in order to allow the CEGAR Core, as well as the designer, to control the way each expression is modeled in YICES. In particular, each non-leaf expression $e = op(e_1,...,e_n)$ is seen as a combinational component with output e, inputs $e_1,...,e_n$, and function op. In turn, annotation is used to indicate to YICES how to model each component:

- The output e can either be represented as a term or a bit-vector. The former modeling is suitable for the EUF and CLU abstractions introduced in Section 4.3, while the latter is used during feasibility checking .

- The operator *op* can be modeled in the SMT formulation using a UF/UP node, a CLU node, or a native bit-vector operation supported by YICES.

As alluded to earlier, the CEGAR Core annotates each expression node prior to passing it to the Solver module, which in turn uses this information to formulate the constraints in YICES. The user has also some control over the way expressions are modeled using this mechanism. The latter can be useful for externally controlling Reveal's initial abstraction $\alpha_0(\cdot)$.

6.1.4 The MUS Extractor

This module is responsible for identifying MUSes from an unsatisfiable formula. Unlike an initial implementation of the system [2], the current implementation uses a modified version of the CAMUS MUS extraction algorithm [38] that works directly with the YICES solver. This eliminates the need to generate a propositional encoding of the abstract formula and leads to significant speedup in MUS generation. It also reduces the number of all possible MUSes in the given conjunction, since including (or excluding) constraints in the MUS is now done at a coarser granularity, allowing CAMUS to scale better. It is worth mentioning that given an unsatisfiable formula, CAMUS can be run in three modes: single-, multiple-, or all-MUS extraction, where the last option is used in most of our experiments.

6.1.5 The CEGAR Core

This component is responsible for coordinating the abstraction, solving, MUS extraction, and refinement processes. It also maintains the persistent lemma database that is accessed across invocations. In each iteration it modifies the lemma database and updates the abstraction for the next iteration. This module is also responsible for integrating user-supplied lemmas into the database.

6.2 A Designer's Perspective

This section briefly demonstrates running Reveal and interpreting its output.

6.2.1 Reveal's Input

The input to Reveal consists of Verilog design files, as well as a *configuration* file containing a set of directives to control Reveal's behavior. The full list of the directives and their functionality and (possible/default) values can be found in Appendix B. The directives can be roughly divided into the following three categories:

- **Algorithmic Behavior**. These directives allow the user to change the default behavior of Reveal's DP-CEGAR algorithm. This includes whether abstraction/refinement is turned on, the type of the abstraction used (EUF or CLU), the type of MUS-based minimization used, and configuring the behavior of the lemma database.
- **Input Specifications**. These directives specify special design signals such as 'clock' and 'property', and additional information on how clocking and unrolling should be modeled (i.e. the number of unrolling cycles, unrolling simplifications used, behavior of clock, etc.). The user can use these parameters also to change the default behavior of the Verilog parser, to make it more compatible with traditional simulation tools (e.g. automatic 0-extension for RHS and LHS expressions that are not size-compatible). Additionally, a number of parameters allow the user to flag certain modules with additional attributes; this includes specifying the top module for multi-module designs, as well as symbolic initialization of memory arrays. Finally, the user can specify the input files to Reveal. This includes names, locations, and types of files for the design and property being checked; aside from Verilog, the user can use the native binary format of the Hardware Relations library.
- **Output Specifications**. These directives control Reveal's output, including the generation of a combinational representation of the initial verification condition, as well

as the final condition (property with lemmas). Reveal supports the following formats for output: Yices, UCLID, Verilog, and BAT.

6.2.2 Reveal's Output and Counterexample Traces

Reveal's output usually consists of three major parts: a preface listing Reveal's current configuration as described in the previous subsection; whether the property holds or is violated; and, in case the property is violated, a counterexample trace. Reveal can also be integrated with a simulation-like graphical back-end interface, as demonstrated in [69]. This section, however, focuses on the textual output of Reveal.

A counterexample trace produced by Reveal includes a list of signals and their corresponding bit-vector values in each cycle (see Figure 6.1). As mentioned earlier, flattening renames design signals based on the hierarchical instantiation of the design's modules. For example, the 'flat signal' pipeline_design1_0$wb_stage_design1_0$result_mux represents the 'hierarchical signal' result_mux that was instantiated in the WB module in pipeline_design1. Finally, it is worth mentioning that Reveal excludes signals and values that are irrelevant to the current bug being diagnosed. This comes mainly in three forms:

- Signals that fall outside the cone of influence of the property being checked, i.e. those signals that do not participate in forming $conc(X)$.
- Signals that fall outside the cone of influence of the path leading to the bug, i.e. a control expression of the form $ite(c, x_1, x_2)$ where $c = 0$ under the current trace (leading to $prop = 0$) prevents signal x_2 from appearing in the trace since its value is irrelevant to the value of this expression for the given value of c.
- Multi-bit signals whose *full* value is not needed to proving $prop = 0$. These mainly include signals involved in bit field selection such as the Instruction Register (see id_ex_IR_design1 in Figure 6.1). In those cases, 'x' is used in the trace to represent a 'don't-care'.

Counterexample Trace:
====================
Cycle 0

---

pipeline_design1_0$wb_stage_design1_0$result_mux
= 0000000000000000000000000000000000000000000000000000000000000001[a]
...
Cycle 1

---

id_ex_IR_design1
= 01000111111111110000010000011111
...
id_ex_IR_design1
= 000xxxxxxxxxxxxxxxxxxxxxxxxxxxxx

[a] The leftmost bit (0) is the vector's most significant bit.

Figure 6.1 A Snapshot from a Counterexample Trace Arising during MIPS Verification

Chapter 7
Experimental Studies

In this section we describe results of applying the abstraction refinement techniques in DP-CEGAR on a number of designs for the purpose of control logic verification. Seven test cases were used to evaluate the effectiveness of those techniques. We also compare our techniques to verification systems being researched and/or developed by others. A summary of the results is presented in Section 7.8.

In the following sections we will classify the various runs of Reveal by a one-, two-, or three-letter code that indicates the abstraction and refinement options used:

- Abstraction options will be labeled B (bit-level, i.e., no abstraction), C (CLU abstraction, see Section 4.3.2), and E (EUF abstraction, see Section 4.3.1).
- Refinement options will be labeled V (negating the violation, as described in Section 4.5) and L (refinement with lemmas, as described in Section 5.1.1).
- For lemma refinement, S will denote refinement with one lemma per iteration, while M will denote refinement with multiple lemmas. For example, the label Reveal(CLM) means CLU abstraction and refinement with multiple lemmas, whereas Reveal(EV) means EUF abstraction and refinement with the negation of the violation.

We compare the performance of Reveal against the following four verification systems:

- UCLID [12][79] which allows modeling of the datapath with abstract terms, and memories with Lambda expressions. Since UCLID does not accept Verilog, we use VAPOR [4] to produce UCLID models.

Table 7.1 Benchmark Statistics

Name	Verilog Lines	Verilog Signals	State Bits
Sorter	79	30	35 to 10^3
ICRAM	153	13	$1.3\text{x}10^5$
OMU	400 to 10^4	40 to 260	$1.0\text{x}10^6$
DLX	$2.4\text{x}10^3$	399	$1.0\text{x}10^{1}1$
RISC16F84	$1.2\text{x}10^3$	169	$1.0\text{x}10^5$
X86	$1.3\text{x}10^4$	$1.0\text{x}10^3$	$5.8\text{x}10^3$

- BAT [41][80] which models memories with set and get functions for reads and writes, respectively, but models the datapath with finite-length bit-vectors. BAT formulations were produced from our verification conditions compiled from the Verilog. We are unaware of any other conversion methods from Verilog to BAT's language.
- VCEGAR [33][81] which performs word-level predicate abstraction on the Verilog input, but does not abstract memory arrays.
- VIS [10][82] which, by default, uses bit-level reachability analysis to verify invariants. It can also be used in two special modes: one that performs bounded model checking of safety properties, and another that performs invariant checking with a CEGAR algorithm based on *hiding registers* [61]. We will denote the default mode by VIS, the BMC mode by VIS(BMC), and the last mode by VIS(AR).

The first six experiments, with design statistics shown in Table tab:stats, were conducted on a 2.2 GHz AMD Opteron processor with 8GB of RAM running Linux, while the last experiment was conducted on 2.0 GHz Intel Xeon processors with 16GB of RAM running Linux. VCEGAR, BAT, and UCLID use the zChaff SAT solver [83] and the SMV model checker [66].

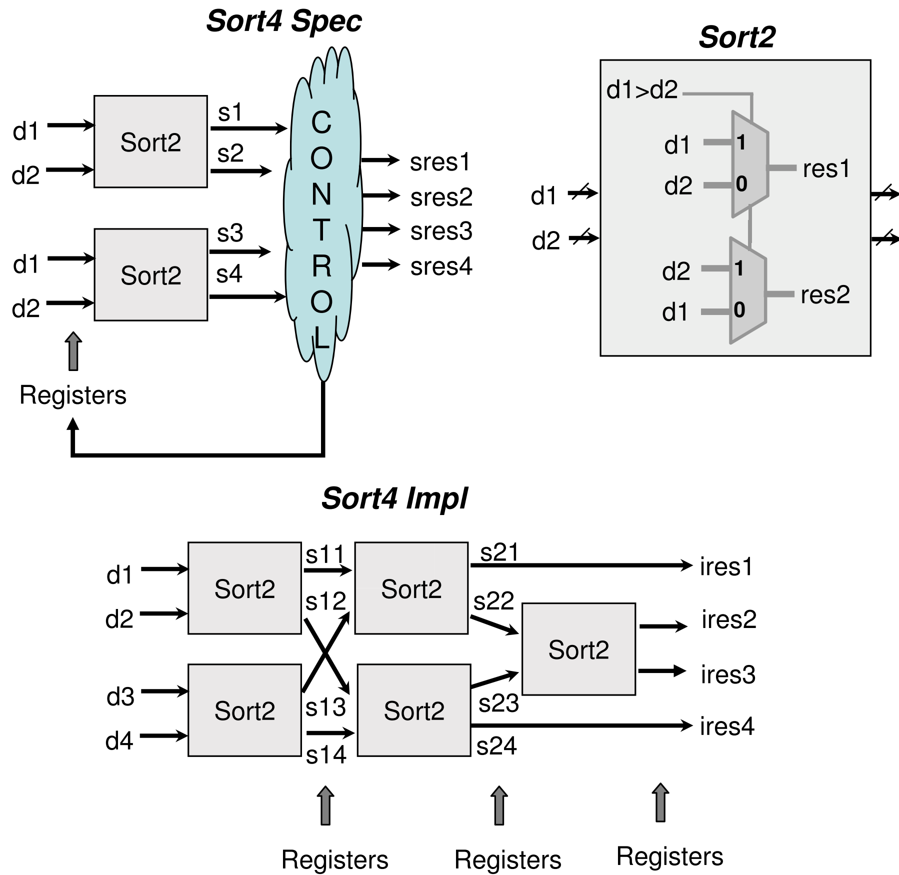


Figure 7.1 Sorter Test Case

7.1 Sorter Case Study

The Sorter design [71] implements two versions of an algorithm that sorts four bit-vectors. It makes use of a Sort2 sub-unit that sorts two bit-vectors. In the first version, five Sort2 sub-units are instantiated and connected serially. The inputs are introduced to the first two sub-units, and the calculation propagates serially towards the outputs. The computation advances through 3 layers of registers, thus requiring three cycles to complete. The second version is based on just two Sort2 sub-units and a controller that uses them to carry out the sorting computation in three cycles as well.

- The effect of datapath abstraction is evident from the performance of Reveal(C) and UCLID, which are oblivious to W. In both cases the abstract model is unaltered when

changing the datapath bit width; thus the time needed to verify the abstract model is constant. Furthermore, the only interaction between the datapath and the control involves bit-vector inequalities, allowing the CLU logic to prove the property without any refinement.

- BAT's performance degrades when increasing W, since the datapath is unabstracted. Nonetheless, BAT's reduction to CNF appears to play an important role in keeping the runtime low.
- VCEGAR takes 6.1 seconds to prove the property for W=2 as it incrementally discovers between 33 and 40 predicates within 58 to 130 iterations. Additionally, the runtime grows exponentially with the width of the datapath. We suspect that the reason behind this is the expense of simulating the abstract counterexample on the concrete design in each refinement iteration, as well as the repeated generation of the abstract model each time a new predicate is added.
- The runtimes of Reveal(B), VIS, and VIS(BMC) degrade rapidly as the bit width is increased. The runtimes of VIS(AR) are similar to VIS and were removed from the graph to avoid clutter.

The property we verified is the equality between corresponding outputs in the two versions. All the bit-vectors in the two units, including the inputs and the outputs, are of bit-width W, which we vary from 2 to 64 to see the effect of the datapath width on the scalability of each tool. Figure 7.2 shows the runtime of each of the verification tools as a function of W, and Table 7.2 shows the number of bits in the concrete verification condition (i.e., $\varphi(\cdot)$) and statistics about the number of the nodes in the abstract verification condition (i.e., $\hat{\varphi}(\cdot)$). The last column, labeled by R, shows the ratio between the number of bits and the number of nodes. The results demonstrate the following trends:

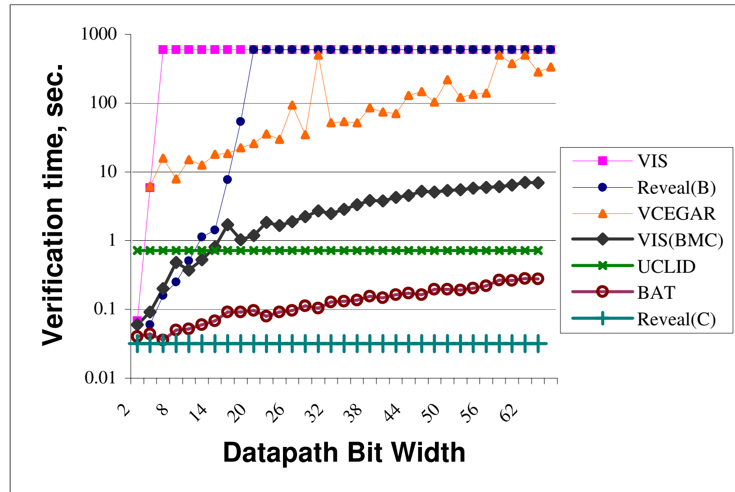


Figure 7.2 Runtime Graphs for Sorter

7.2 Instruction Cache RAM Case Study

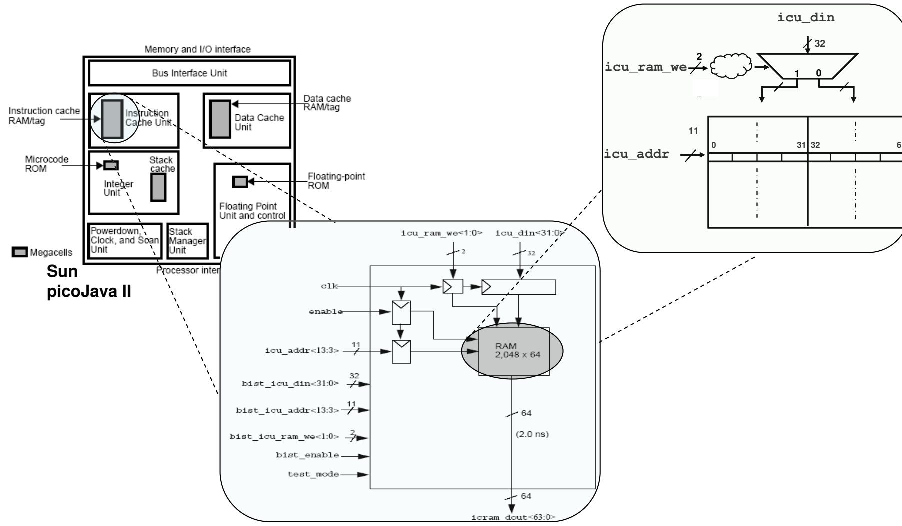
The Instruction Cache RAM (ICRAM) test case [72] is obtained from the publicly available Verilog description of the Sun PicoJava II Microprocessor [67]. This unit includes a memory array of 16K 8-bit words, 32-bit input and output data ports, and single-bit control signals to trigger certain operations in the cache such as reading, writing, BIST testing, and switching to "power down" mode. The ICRAM interacts with the Instruction Cache Unit which manages the instructions tags and buffers for the entire microprocessor.

The address space of the ICRAM is divided into two "banks," distinguished by a single bit in the address register. A write operation takes an address signal adr[13:3], a data signal di[31:0], and control signals selecting the destination bank b$\in\{0,1\}$. The memory update for write(adr,di,b) is:

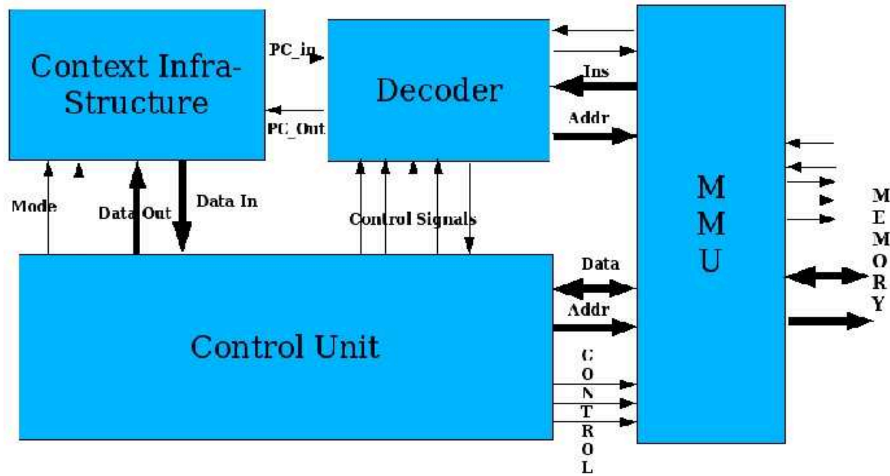
mem[adr,b,00]<=d[31:24]

mem[adr,b,01]<=d[23:16]

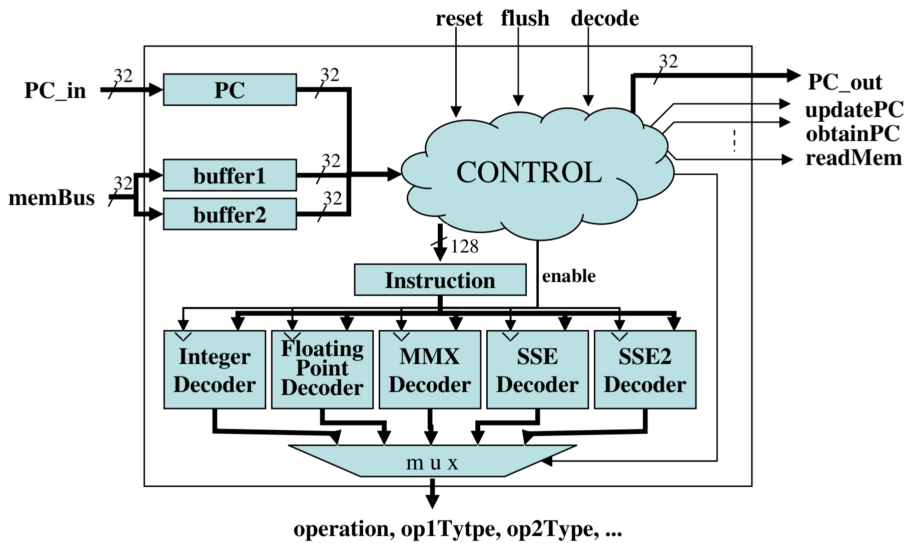
mem[adr,b,10]<=d[15:8]



(a) ICRAM Design



(b) X86 Design



(c) X86 Decoder

Figure 7.3 ICRAM and X86 Testcases

Table 7.2 Verification Condition Stats

Test	$E \rightarrow p$	$A \rightarrow p$					R
	Bits	Terms	Booleans	UFs	UPs	Overall	
Sorter, W=8	127	14	12	0	0	26	5.08
Sorter, W=16	249	14	12	0	0	26	9.96
Sorter, W=32	473	14	12	0	0	26	18.9
Sorter, W=64	921	14	12	0	0	26	36.8
ICRAM	287	31	48	9	2	90	3.12
OMU, K=16	1346	67	275	2	0	344	3.91
OMU, K=32	3154	131	1059	2	0	1192	2.65
OMU, K=64	8306	259	4163	2	0	4524	1.88
OMU, K=128	2.5×10^4	515	1.7×10^4	2	0	1.7×10^4	1.47

mem[adr,b,11]<=d[7:0]

The ICRAM has been formally verified by VCEGAR [33] and BAT [41]. The property verified is that given an arbitrary initial memory array, performing a write(adr,di,0), then performing a read from address adr,001, will yield a value that is equal to di[23:16].

We verified this example with Reveal(C), Reveal(B), BAT, and UCLID. The runtimes are 30ms, 38ms, 50ms, and 92ms, respectively. This result is counterintuitive given that the original design has 2^{17} state bits. The efficiency of these methods stems primarily from the reduction obtained by memory abstraction; as shown in Table 7.2, both the concrete and the abstract verification conditions are very small despite the large state space. Moreover, due to the simple interaction between the control and datapath, the abstraction in UCLID and Reveal(C) is sound and complete. Therefore, refinement is not triggered.

Left unabstracted, the memory array causes VCEGAR and VIS to encounter "vertical" state explosion. VCEGAR's runtime was shown in [33] and [41] to grow exponentially with the memory size. Likewise, VIS times out for this example. In particular, the verification in VIS begins with converting any n-word by m-bit memory into $n \cdot m$ single-bit registers regardless of the property being verified. "Flattening" the memory in this way also leads to loss of the structural correlation between the memory registers, which can

otherwise be used by the model checker during verification.

7.3 Out-of-Order Memory Updates Case Study

The Out-of-Order Memory Updates example [73] (OMU) has been previously introduced in [41] to demonstrate the effectiveness of memory abstraction for RTL verification. The design instantiates an array of 65K 16-bit words, which can be read from or written to via designated signals.

The design is verified by simulating two sequences of write operations on the memory array. The initial memory M is modified by a sequence of K writes to locations $A, A+1, A+2, \ldots, A+K-1$, with the data words $D_1, D_2, \ldots, D_K$, respectively, resulting in memory $M1$. Independently, a second sequence of writes is performed on M in locations $A+K-1, A+K-2, \ldots, A$, with the data words, $D_K, D_{K-1}, \ldots, D_1$, respectively, resulting in memory $M2$. Since the addresses for the write operations are mutually distinct, the ordering of the writes does not affect the final state of the memory. In particular, the content of location A in both $M1$ and $M2$ is equal. A second, more generic, property is verified by simulating a similar sequence of writes to distinct locations $A_1, A_2 \ldots, A_K$. In other words, we allow the addresses to be arbitrary, albeit mutually dis-equal.

We compared Reveal(C), Reveal(B), BAT, and UCLID on these two properties, while varying K over $\{16,32,64,128\}$. The runtimes are plotted in Figure 7.4 (page 79) on a logarithmic scale[1]. Similarly to the ICRAM case, the effect of modeling the memory is evident in this example. In particular,

- Reveal(C) scales well on both properties, taking less than 3 seconds for all the values of K. This is attributed to the memory abstraction via Lambda expressions [12]. Refinement was not triggered since the datapath/control interactions are exclusive to (dis-) equalities.

[1] Dashed and solid lines correspond to the first and second properties, respectively. VIS and VCEGAR were omitted to avoid clutter.

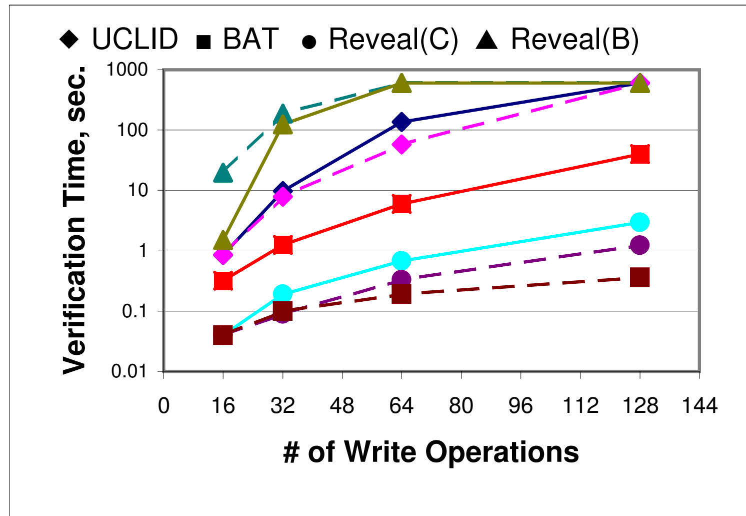


Figure 7.4 Runtime Graphs for OMU

- BAT appears to be sensitive to the pattern of memory writes; proving the property for arbitrary addresses is two orders of magnitude slower than for consecutive addresses.
- UCLID is two orders of magnitude slower than BAT and Reveal(C) on both properties. Despite its memory and datapath abstractions, its reduction to CNF [41] is significantly slower in proving the property on the abstract model.
- Reveal(B) clearly demonstrates the state explosion problem, as the runtime rapidly worsens when increasing K.
- As with the ICRAM case, VCEGAR's runtime was shown in [41] to grow exponentially in the number of writes to memory. VIS times out on this example for any number of writes. The lack of memory abstraction hinders both.

7.4 DLX Case Study

DLX [74] is a 32-bit RISC microprocessor [30]. Its salient features include a 32-bit address space with separate instruction and data memories, a 32-word register file with two read ports and one write port, and 38 op-codes for arithmetic, logical, and control operations.

Our case study involved comparing two versions of DLX, both written in Verilog 95 [54]. The first version, which we will refer to as *DLXSpec*, is a single-cycle implementation of the instruction set architecture (ISA) and serves as the architectural specification of the microprocessor. The second version, labeled *DLXImpl*, is a standard 5-stage pipelined design consisting of instruction fetch, instruction decode, instruction execute, memory access, and write-back stages.

Starting *DLXSpec* and *DLXImpl* from their reset states, the property we checked for was equivalence of corresponding state elements (register and memory locations) after a bounded number of execution cycles. Specifically, let E_i^S and E_j^I denote the values of two corresponding state elements from the specification and implementation after i and j cycles from reset, respectively. These two elements would, then, be considered equivalent if:

$$(E_1^S = E_1^I) \vee (E_1^S = E_2^I) \vee \ldots \vee (E_1^S = E_5^I)$$

To compare the various abstraction and refinement options in Reveal and to demonstrate its ability to (dis-)prove properties, we verified a number of (buggy and bug-free) variations of the design. We focus on $E \doteq PC$ here, but similar verification can be used for other state elements. The buggy versions were obtained by injecting errors in the RTL of DLXImpl. These variations are as follows:

- D1 is a bug-free $DLXSpec$ and $DLXImpl$.
- D2 is a buggy $DLXImpl$ wherein the pipeline ‘Stall’ control signal is stuck at 1.
- D3 is a buggy $DLXImpl$ wherein the address of the ‘jump’ instruction is calculated incorrectly.

Table 7.3 contains runtime statistics for each mode of Reveal. Columns labeled T, I, and L, describe, respectively, runtime (seconds), number of iterations, and total number

Table 7.3 Verification Results for DLX

	CV[a]		ELS		CLS		ELM				CLM				B
	T	I	T	I	T	I	T	A	I	L	T	A	I	L	T
D1	>600	>1507	1.92	9	1.8	8	**0.6**	39	4	8	1.0	27	6	12	>600
D2	**0.11**	1	0.15	1	**0.12**	1	**0.11**	3	1	0	**0.1**	7	1	0	0.21
D3	3.16	45	2.22	11	1.16	5	**1.13**	23	3	5	**1.1**	25	4	8	6.7

[a]The notation is explained in the beginning of this chapter.

of refinement lemmas (when applicable). The columns labeled A show the ratio of the runtime of verifying the abstract model to the total runtime as a percentage. Finally, the smallest runtime is emphasized in each row; there can be multiple in each row when the difference is insignificant.

The performance of the various options in Reveal demonstrate the role of automatic refinement. Since the control and the datapath in this design are intermixed, refinement is needed to "recover" facts that were lost in the course of the abstraction, yet are relevant to (dis-)proving the property. To shed some light on the types of lemmas discovered during this process, we traced the source of these lemmas back to the original Verilog code. Most of these lemmas were related to the pipeline registers and control logic in DLXImpl. For instance, the lemma (IR3=32'd0)→(IR3[31:26]≠6'd4), which states that it's not possible to extract a non-zero field from a zero bit vector, was traced to the following code segment involving IR3:

define BEQ 4

define op 31:26

initial IR3 = 32'd0;

case IR3[‘op] ‘BEQ: ...

In this case, the initial abstraction lost the fact that IR3[31:26] can not be equal to 4

(i.e., the opcode of BEQ) when it is actually holding a NOP instruction (i.e., with opcode 0), and it found a spurious counterexample that executed the BEQ instruction.

Upon closer examination, we found that *DLXImpl* consists mainly of a datapath that is responsible for computing values for the PC and memory to be committed, and control logic that orchestrates the pipeline. Furthermore, the datapath in *DLXImpl* is very similar, and in most cases identical, to the datapath in *DLXSpec*. As a result, refinement only affects those portions of the design involving interactions between the datapath and control logic in *DLXImpl*.

Table 7.3 also shows that the use of lemmas for refinement (modes ELS, CLS, ELM, and CLM) is far superior to using the violation (mode CV). Also, using multiple lemmas in each refinement (modes CLM and ELM) outperforms refinement with a single lemma at a time (modes ELS and CLS).

Surprisingly, Reveal(B) is able to terminate on the buggy versions of the design. This is attributed to the ability of the BV solver in YICES to efficiently find a satisfying assignment to equation 4.3. The rest of the case studies in this thesis confirm that proving the unsatisfiability of this equation is intractable with Reveal(B), while proving its satisfiability may be tractable in some cases, though not all.

In order to compare the performance of YICES and UCLID in solving the abstract formula, we generate the expression: $(conc \rightarrow prop) \vee \bigvee_i lemma_i$ which represents the final "refined" verification condition created in Reveal. This expression is dumped as a Verilog word-level combinational circuit, and VAPOR is then used to generate its corresponding UCLID model. UCLID spends two orders of magnitude more time than the time spent by Reveal in solving the abstract formula. We observed a similar trend in the rest of the test cases.

Finally, we ran VIS and VCEGAR on this design. VIS was unable to create a netlist due to what we believe is an internal error in the tool. Regardless, we do not think that VIS could verify this design due to its large memory arrays. VCEGAR processed the input but

timed out at 600 seconds.

7.5 RISC16F84 Case Study

This design is an implementation [75] of the Risc16F84 microcontroller [78]. It has a 213x14-bit instruction memory, a 29x8-bit data memory, 34 op-codes, and a 4-stage pipeline. We denote the implementation and specification by *OCImpl* and *OCSpec* respectively. *OCImpl* processes one instruction every four cycles, while *OCSpec* needs one cycle to process each instruction. The equivalence criterion in this case is

$$\bigwedge_j (I_0^j = S_0^j) \rightarrow \bigwedge_j (I_4^j = S_1^j),$$

where I_i^j and S_i^j denote the state of the jth state element in *OCImpl* and *OCSpec*, respectively, after i cycles of execution. In essence, this is an inductive criterion: given equal state elements in the current cycle, it requires equal state elements after processing a single instruction.

Reveal was able to discover a genuine bug in this design. The following Verilog code in *OCImpl* uses a *floating* signal c_in as the carry-in bit to a 8-bit addition operation.

```
// risc16f84_lite.v
reg c_in; // line 223
add_node,temp <= {1'b0,aluimpl_reg,1'b1}+{1'b0,aluinp2_reg,c_in}; // line 519
```

OCSpec, on the other hand, performs addition without any carry-in bit. Reveal thus produces a counterexample showing the deviation, with c_in assigned to 1. The unit designer acknowledged this problem, and asserted that the simulation carried out for this design assumed c_in=0.

Table 7.4 Verification Results for RISC16F84

	CV[a]		ELS		CLS		ELM				CLM				B
	T	I	T	I	T	I	T	A	I	L	T	A	I	L	T
R1	>600	>1767	>600	>1204	>600	>1085	257	.7	93	185	**148**	.8	68	170	209
R2	**0.79**	8	56	20	>600	>1881	72	1	44	13	40	1.1	33	39	15.2
R3	115	654	50	123	121	311	**2.6**	.6	5	15	27.3	0.2	40	73	11.6

[a] The notation is explained in the beginning of this chapter.

Tables 7.4 contains runtime results for three versions of this design:

- R1 is a bug-free *OCImpl* and *OCSpec*.
- R2 is a buggy *OCImpl* with the aforementioned bug, i.e. a floating 'carry-in' signal for addition.
- R3 is a buggy *OCImpl* wherein 'aluout_zero_node' is stuck at 1.

In these results we observe the following:

- Refinement with lemmas is superior to refinement with the violation. Furthermore, the use of multiple lemmas for refinement is crucial for verifying version R1.
- The verification condition here is relatively small despite the huge memory embedded in the RISC16F84 design. This is attributed to memory abstraction discussed in previous sections.
- The verification of the bug-free version (R1) with Reveal(B) terminates after 209 seconds. It also terminates rapidly on the 2 buggy versions. This makes its performance comparable with Reveal(C) and Reveal(E). As we saw in previous sections, the runtime of Reveal(B) grows exponentially with the number of bits in the concrete verification condition. On the other hand, the performance of Reveal(C) and Reveal(E) depends on the number of nodes in the verification condition as well as the control/datapath intermix.
- The R2 case shows an interesting outlier, in which Reveal(CV) is significantly faster than any version that refines with lemmas. This is due to the heuristic nature of the

satisfiability search for finding a bug. Any search, regardless of the refinement used, could “get lucky” and reach a bug early in this way, though only rarely.

- An analysis of the lemmas discovered in all variations of this test case reveals that most of the spurious counterexamples are due to the *variable opcode width* feature, wherein the opcode field can be K-bits wide for any $K \in \{2,3,4,5,6,7,14\}$. For instance, the opcode of the *goto* instruction is IR[13:11]=3’b101, while the opcode for addlw is IR[13:9]=5’b11111. The encoding guarantees that only one opcode is active at any given time. This information is lost when abstracting the bit-vector extraction operation. This results in the occurrence of lemmas of the form $(IR[13:k_1] = v_1) \rightarrow (IR[13:k_2] \neq v_2)$ for values v_1, v_2 and distinct indices $k_1, k_2 \in K$.
- On this example, UCLID timed-out after 600 seconds for R1, and is two orders of magnitude slower than YICES on R2 and R3. VCEGAR runs out of memory after 370 seconds, and VIS was not able to process this design since it does not support blocking assignments, which are used throughout the Verilog description. We believe that VIS would otherwise encounter an additional obstacle with the large memories.

7.6 X86 Case Study

The X86 design [76] is an open source RTL Verilog model developed at IIT, Madras that implements Intel’s IA-32 ISA [77]. The design contains four high-level modules. The *Decoder* module, which is the main focus of our verification effort, is responsible for fetching an instruction prefix from the memory, finding the total length of the instruction, fetching and decoding the rest of the instruction, and providing the result to the *Control* module. The top module of the Decoder instantiates the fetching unit, the instruction length find unit, and six decoding units, which correspond to six instruction types that exist in the x86 architecture and its extensions, namely Integer, Floating Point, MMX, SSE, SSE2, and SSE3. Each decoding unit has an enable signal that orchestrates its operation with the

Decoder top module.

Upon reset, the Decoder fetches the PC and the corresponding instruction from memory. We verified the property that the Decoder activates the corresponding decode unit when the instruction is confined to a set of 6 Integer and Floating Point op-codes as follows:

$$(opcode \in \{CMP, JMP, MOV, FADD, FCMOV, FINIT\}) \rightarrow$$
$$((opcode \in \{CMP, JMP, MOV\}) \leftrightarrow en_{Integer}) \wedge$$
$$((opcode \in \{FADD, FCMOV, FINIT\}) \leftrightarrow en_{FloatingPoint})$$

When the verification was invoked in Reveal, the tool was able to discover a coding problem in the design. In particular, the RTL description includes the code

```
// sse3Decoder.v
op2 = 32d0; // line 55
if (...) // line 185
   op2[16:0] = instrSeq[31:16]; // line 188
```

which uses a blocking assignment to initialize the signal op2, and then extracts a 16-bit displacement value from the instruction stream and assigns it to a 17-bit register. Most synthesis tools will zero-extend the RHS expression to make the sizes consistent, in which case the resulting model is still correct. Nonetheless, such an error may indicate additional problems in other units of the design. We have notified the unit designers about this problem, and we modified the Verilog to eliminate the problem for the later experiments.

Similarly to the previous two test cases, we compared the performance of Reveal on two buggy versions and one bug-free version as follows:

- X1 is a bug-free X86 design and property.
- X2 is a buggy version wherein property swaps ‘enable’ signals for the Int and FP

Table 7.5 Verification Results for X86

	CV[a]		ELS		CLS		ELM				CLM				B
	T	I	T	I	T	I	T	A	I	L	T	A	I	L	T
X1	>600	>388	>600	>1158	>600	>945	**36.5**	31	40	104	60.4	59	19	96	>600
X2	>600	>461	>600	>1062	>600	>1046	**30.5**	32	78	161	103	63	24	86	>600
X3	**1.98**	2	1.95	2	**1.96**	2	**2.0**	6	2	6	2.1	6	1	0	2.72
X4	>600	>308	>600	>847	>600	>1252	**23**	48	12	41	58.7	74	7	43	>600

[a] The notation is explained in the beginning of this chapter.

units.

- X3 is a buggy design wherein the ‘opcode’ for ‘CMP’ activates the FP unit instead of Int unit.

The runtime results are included in Table 7.5. These results reassert the importance of refinement with multiple lemmas. A notable phenomenon in this case is that Reveal(C) converges significantly faster than Reveal(E) in terms of refinement iterations. This is attributed to the heavy use of counters in the FSM of the X86 decoder. Along these lines, note that the number of lemmas accumulated in Reveal(C) is much smaller than in Reveal(E). On the other hand, Reveal(C) spends more time verifying the abstract model, almost twice as much as Reveal(E), despite Reveal(C)s smaller number of refinement iterations.

To further assess the effect of the lemma database on the convergence of the algorithm, we ran Reveal(C) on a version that combines the three bugs present in X2, X3 and X4. This was an iterative session, in which Reveal was re-invoked after correcting each reported bug. We tested Reveal in two modes: a mode in which learned lemmas are discarded after each run and a mode in which learned lemmas are saved and used across runs. The total runtime for the first mode was 232 seconds, whereas the runtime in the second mode was 166 seconds, a 40% improvement in speed. This confirmed our conjecture that lemmas discovered in one verification run can be profitably used in subsequent runs. The verification of real-life designs involves tens to hundreds of invocations of the tool, thus a significantly larger speedup could be seen in practice.

UCLID exhausts available memory during its CNF encoding stage on most of the variations of this design after approximately 250s. VIS cannot process the input Verilog due to blocking assignments, and VCEGAR halted due to an internal error after parsing.

7.7 MIPS Case Study

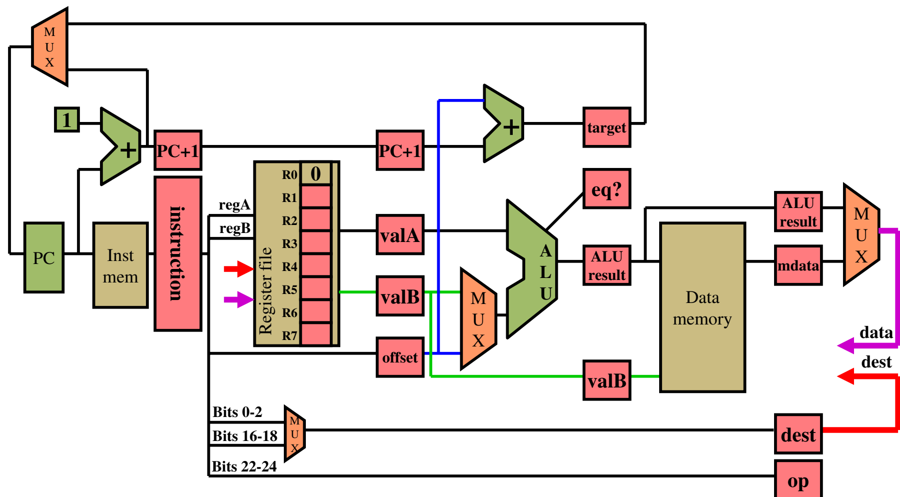
MIPS [70] is a 64-bit microprocessor implementing the Alpha ISA. Aside from a wider datapath, it differs from the DLX design that was introduced in Section 7.4 mainly in the fact that it follows the von Neumann architecture rather than the Harvard architecture used in the DLX; the instruction and data memories are unified, and thus the microprocessor can either read instructions or read/write data from/to the memory on the same bus.

Unlike its predecessors, this case study

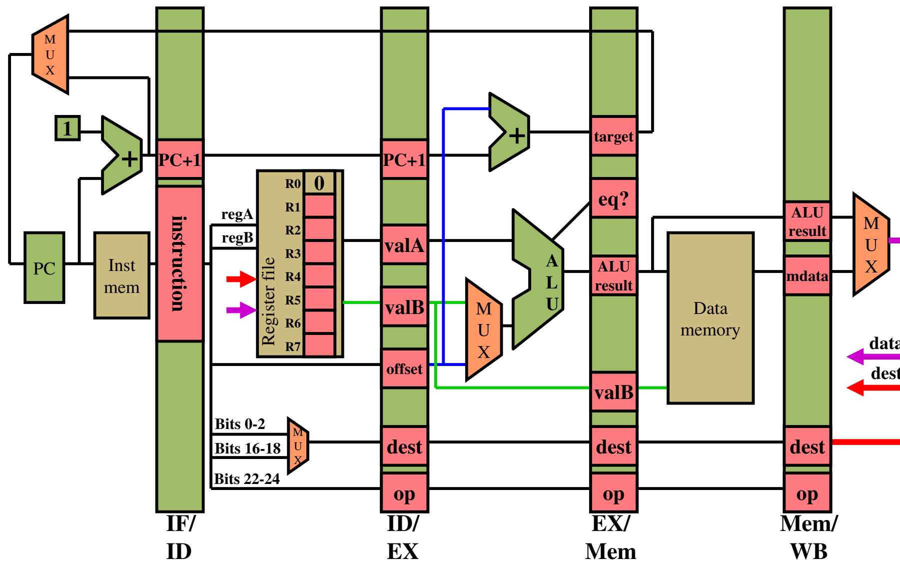
- demonstrates Reveal's ease of use, as it is invoked by 21 different designers on their own variations of the MIPS;
- showcases the tool's ability to discover real and subtle control bugs in pipelined implementations of the MIPS;
- and sheds some light on the scalability of Reveal for proving correctness or discovering bugs.

Three variations of the MIPS design were involved in the verification effort. The first version, called *MIPSSpec* and described in Figure 7.5(a), implements the ISA with a single-cycle-per-instruction design. The second version, called *MIPSBubble* and shown in Figure 7.5(b), is a simplified version of the 5-stage pipeline that loads one instruction every 5 cycles, to allow trivial resolution of hazards; after each instruction, 4 NOPs are pushed into the pipeline for that purpose. The third version, called *MIPSPipe* (Figure 7.6(a)), is a full fledged implementation of the pipeline with stalling and forwarding logic.

The verification of MIPS was done in two experiments. The first experiment, referred to as *Bubble-to-Bubble*, was initially expected to produce a trivial result. However, Re-

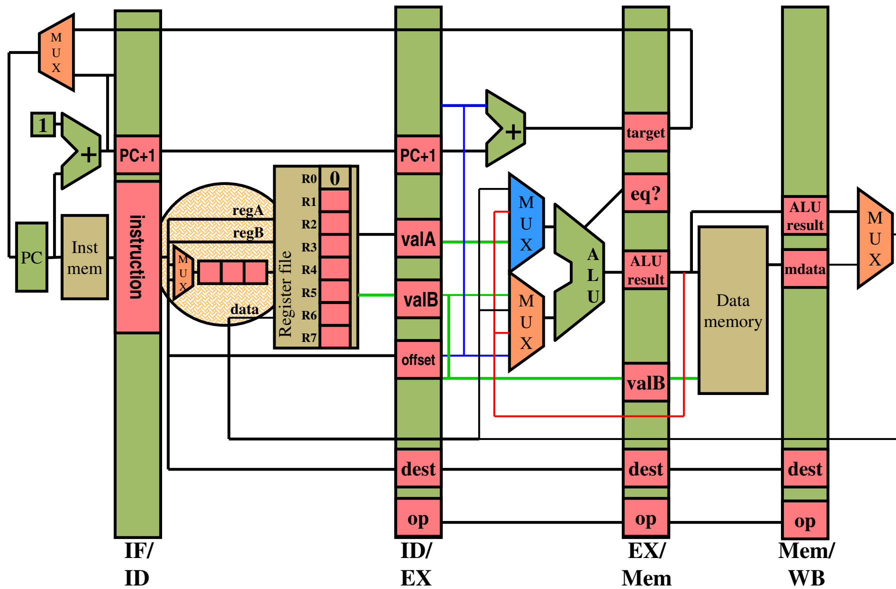


(a) Specification Design (*MIPSSpec*)

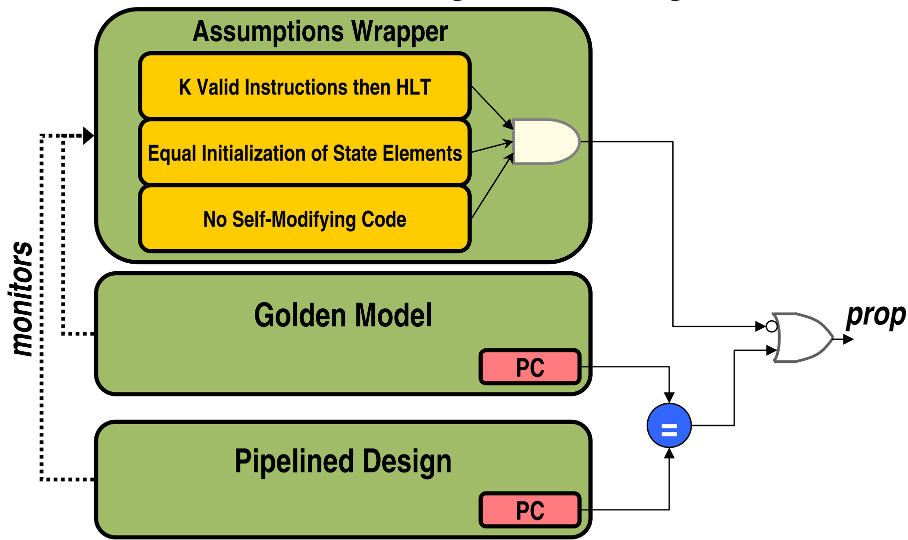


(b) NOP-based Pipeline Design (*MIPSBubble*)

Figure 7.5 MIPS Specification and a Naïvely Pipelined Implementation



(a) A Fully Pipelined MIPS Design (*MIPSPipe*) with Hazard Detection, Stalling, and Forwarding



(b) The Full Circuit Combining *MIPSSpec*, *MIPSPipe*, and the Property Description

Figure 7.6 MIPS Full Pipeline and Equivalence Circuitry

veal discovered a counterintuitive error that was present in this design, which was used for years by computer architecture students across many universities. In this experiment, two identical *MIPSBubble* designs, dubbed 'design1' and 'design2', were compared for equivalence, and were found by Reveal to be non-equivalent, indicating non-determinism in the execution of the microprocessor.

Visible state elements, i.e. PC, memory, and register files, were initialized similarly in both designs. Additionally, both versions were simulated for 5 cycles, to allow fetching, executing, and retiring a single instruction. Surprisingly, Reveal discovered that the retired values are not similar, indicating that non-determinism is present in the pipeline, due to improper initialization. This was attributed to the *write_enable* signal of the register file, which was active at the first positive clock edge despite the reset operation. As a result, a write operation took place at the first cycle without an explicit instruction through the pipeline dictating so.

The counterexample trace indicated that a 'conditional jump' instruction triggers this problem. Since programs would normally avoid loading or branching on a dirty value from the register file (i.e. without previously writing to that location), this error is not triggered by most programs that follow usual programming semantics. However, this error shows that unintended behavior was introduced to the microprocessor, which could not have been discovered without the use of formal verification with Reveal.

The next few paragraphs explain Reveal's counterexample trace indicating the aforementioned bug. During our explanation, we will refer to the events listed in Table 7.6 and the full counterexample trace in Appendix C.

1. **PC Initialization:** In cycle 0, both instances of the design are reset on the positive edge of the clock, forcing their PC to get the value zero.
2. **Memories Initialization:** The content of the register file and memories are initially forced to be identical across the two designs. As shown in Table 7.6, this is done by directing Reveal to replace all memory arrays in cycle 0 called 'memarray_*' with

Table 7.6 MIPS Bubble-to-Bubble Bug Reference

Event	Verilog Fragment	File[a]	Line(s)
1	PC_reg <= 64’0;	if_stage_*.v	76
2	memarray_* memarray regf_* regarray	*.mem_map	1-4
3	assign reg_wr_en_out = (mem_wb_dest_reg_idx!=‘ZERO_REG);	wb_stage_*.v	59
4	if(op_code == 6’h38) opa_select = ‘ALU_OPA_IS_NPC; opb_select = ‘ALU_OPB_IS_BR_DISP; alu_func = ‘ALU_ADDQ; cond_branch = ‘TRUE;	id_stage_*.v	253,256 257,258 276
5	2’b00: cond = (opa[0] == 1’b0); wire [63:0] br_disp = {{41{id_ex_IR[20]}},id_ex_IR[20:0], 2’b00};	ex_stage_*.v	118,172

[a] ‘*’ is used to indicate both ‘design1’ and ‘design2’.

‘memarray’, and consequently forcing both memory arrays (from ‘design1’ and ‘design2’) to be equal. The same is applied on the register file. These directives are added to the memory mapping files (using ‘mem_map’), as explained in Appendix B.2.2.

3. **Erroneous RF Modification:** Erroneously, the ‘write_en’ signal of the register file assigned in the WB stage is active during ‘reset’. As illustrated in Table 7.6, as well as Appendix C lines 63, 64, 69, and 70, Reveal chooses an initial value of 5’b00000[2] for the destination register, which leads to an active ‘write_enable’, which results in writing value 64’d1 to r0.
4. **Conditional Branch Instruction:** As shown in Appendix C lines 204 and 205, the ‘IR’ has MSBs of 6’b111000=6’h38. The decoder unit interprets that as a conditional branch, which, in turn, directs the ALU to calculate the next value of the PC based on NPC+displacement if the condition holds.
5. **NPC Calculation:** As shown in Appendix C lines 312-317, 322, and 323, the values of the ‘IR’ and ‘rega’ in design2 during ‘EX’ stage leads to the addition of

[2] The ‘ZERO_REG’ is register 31.

64’hFFFFFFFC to ‘NPC’, whose value is 64’h4, leading to changing the ‘PC’ to 64’d0 in ‘design2’, while ‘design1’ has a PC of 64’d4. The source of the deviation is the value of r0 which differs in both designs, due to the erroneous RF modification in cycle 0 as explained earlier; while r0 in ‘design1’ is 64’d1, leading to ‘cond’=0 (i.e. jump is not taken), r0 is 64’d0 in ‘design2’, leading to ‘cond’=1 (i.e. jump is taken).

6. **Final PC Deviation:** As shown in Appendix C lines 464-467, the final ‘PC’ values in ‘design1’ and ‘design2’ are, respectively, 64’d4 and 64’d0, leading to falsifying the equivalence.
7. **Bug Fix:** To fix this bug, the statement in ‘wb_stage_*:line 59’ is modified to: assign reg_wr_en_out = (mem_wb_dest_reg_idx!=‘ZERO_REG) && !reset;

In the second experiment, *MIPSSpec* is compared to *MIPSPipe* for *k* instructions after reset, including a HALT instruction at the end of the stream (see Figure 7.6(b)). The pipelines were simulated with *k* arbitrary instructions that follow these assumptions:

- The instruction currently in the WB stage is a HALT. This was enforced with a special flag triggered by the HALT instruction in the WB stage.
- No HALT instruction was previously encountered in the stream.
- No illegal instruction was previously encountered in the stream.
- No self-modifying code is performed. This was enforced by assuming that instruction accesses are done to addresses less than 100, and data accesses are done to addresses more than 200.

This experiment was done through an undergraduate Computer Architecture Class project at the University of Michigan, with the assistance of the class instructors, Steven Pelley and Prof. Thomas Wenisch. Over 40 students were required to modify *MIPSBubble* by adding stalling and forwarding logic that resolves dependency hazards. They were then asked to formally verify the resulting *MIPSPipe* against *MIPSSpec* using Reveal. 20 students used Reveal to verify their designs, 16 of which submitted their Verilog designs

Table 7.7 MIPS Spec-to-Impl Student Verification Results

Student	Cycles	k	Time	Property	Lemmas
1	12	2.5	136	Pass	236
2	13	3	3541	Pass	4916
3	5	0	0.04	Pass	0
4	11	2	64	Fail	30
5	13	3	507	Fail	1262
6	13	3	1214	Pass	5611
7	13	3	>3600	N/A	N/A
8	13	3	>3600	N/A	N/A
9	12	2.5	113	Pass	179
10	11	2	89	Fail	190
11	11	2	46	Fail	248
12	11	2	1	Pass	0
13	11	2	0.04	Pass	0
14	12	2.5	10	Fail	352
15	11	2	3	Fail	0
16	12	2.5	120	Pass	189

alongside verification results from valid Reveal invokations[3].

The verification results of the students are collectively given in Table 7.7. The 'Cycles' column shows the maximal cycle number for which the student verified her MIPS design. A cycle corresponds to a positive or negative edge of the clock. The next column shows the number of instructions (k) in the corresponding instruction stream. Since each instruction needs two cycles to advance in the pipeline (two clock edges), $k = 2.5$ means that two instructions fully retired, and the third instruction needs half a clock period to retire. The last three columns show the verification run time in seconds, the result of the equivalence checking, and the number of total lemmas aggregated in the database.

The results show that

- Reveal is suitable for both finding design bugs (e.g. the case for students 4, 5, 10, 11, 14, and 15), or proving design correctness.
- With a time-out of 1 hour, most students were able to run Reveal to completion.

[3]Remaining students did not expose their designs to us, or failed to supply Reveal with valid Verilog input.

- Reveal learns 1 to 7 lemmas per second, and tends to require more lemmas on average when the property fails than when it passes.
- Reveal requires different number of lemmas to converge (and in turn the verification time is affected) across various designs with similar cycle number and verification result. For example, student no. 2 requires 3 times more lemmas and run time than student no. 6, although both run with cycles=13 and with a correct design. This indicates that the design of student no. 3 involves more datapath/control interactions that had to be fixed using refinement.

The case of student no. 16 is particularly worth noting. Gong Yifan Yang (Yifan in what follows) discovered two problems in his MIPS pipeline with Reveal, both of which are related to his forwarding logic. The Verilog signals 'fwd_check_EX' and 'fwd_check_MEM' are used throughout his design to indicate forwarding respectively from the 'EX' and 'MEM' stages into the 'ID' stage. These 5-bit variables take on values ranging from 5'd0 to 5'd31, indicating the register index that should be forwarded. Yifan's *MIPSPipe* had two bugs:

- These signals were not initialized in one version of his design, leading to forwarding from 'EX' and 'MEM' stages prior to loading meaningful instructions into them.
- These signals were erroneously assigned value 5'd30 upon reset or a taken branch. While Yifan's intention was to assign them to 5'd31 (the 'zero register'), which is treated as 'no forwarding', Yifan's design actually triggered forwarding (immediately after reset and after a taken branch) from register 30.

For brevity, we will explain the second bug since it subsumes the first. Reveal's full counterexample trace, as produced by Reveal during Yifan's verification effort, is given in Appendix D. Lines 150-153 and 442-445 show faulty forwarding from both 'EX' and 'MEM' after loading the first instruction, and faulty forwarding from 'MEM' after loading the second instruction. In general, forwarding affects instructions that read values from

the register file and produce results *only based on those values that were previously read*. Therefore, instruction sequences that are affected from erroneous forwarding upon initialization, such as the one encountered, remain absent from real life scenarios. Therefore, Yifan's design passed all simulation test cases provided with the project.

However, erroneously activating forwarding *after a taken branch* is a *real* bug for sequences where a conditional jump is followed by instructions that read from r30. This is true since the faulty forwarding forces the pipeline to ignore the actual value of r30 after the branch.

7.8 Experimental Observations

This section generalizes the experimental results introduced earlier, and presents our conclusions regarding the merits of the verification approach, its applicability to complex test cases, its drawbacks, and directions for improvement. The section is divided into a number of themes, each of which puts the results in a different perspsective.

7.8.1 Datapath and Memory Abstraction

The merits of datapath and memory abstraction is evident in most of the test cases. In the Sorter test case, Reveal reduces the verification task to performing validity checking of EUF or CLU formulas that are oblivious to the size of the datapath, contrary to most other verification tools. The OMU and ICRAM test cases also show the effectiveness of memory abstraction, as well as the merits of Lambda-based memory abstraction compared to other memory abstraction methods. Finally, the rest of the test cases show that datapath abstraction is essential to scalability, without which verification is rendered intractable.

This anticipated result has two interesting caveats. First, the use of counting arithmetic in CLU does not necessarily speed up the overall verification. While convergence tends to be faster, the SMT solver spends more time solving the abstract formula. Second, on

Table 7.8 Verification Results for DLX, RISC16F84, and X86

	CV[a]	ELS	CLS	ELM	CLM	B
D1	>600	1.92	1.8	**0.6**	1.0	>600
D2	**0.11**	0.15	**0.12**	**0.11**	**0.1**	0.21
D3	3.16	2.22	1.16	**1.13**	**1.1**	6.7
R1	>600	>600	>600	257	**148**	209
R2	**0.79**	56	>600	72	40	15.2
R3	115	50	121	**2.6**	27.3	11.6
X1	>600	>600	>600	**36.5**	60.4	>600
X2	>600	>600	>600	**30.5**	103	>600
X3	**1.98**	1.95	**1.96**	**2.0**	2.1	2.72
X4	>600	>600	>600	**23**	58.7	>600

[a] The notation is explained in the beginning of this chapter.

a number of occasions, especially buggy versions of the design or specification, leaving the formula unabstracted (i.e. at the bit level) allowed Reveal to terminate quickly. This suggests that adaptive and partial abstraction of the datapath may combine the merits of both methods.

7.8.2 Refinement Trade-Offs

A rapidly converging refinement back-end is essential to the practicality of our approach. Table 7.8 shows the runtime of Reveal (in seconds) on the DLX, RISC16F84, and X86, based on the refinement mode used. In general, the use of lemma refinement, with multiple lemmas in each iteration, outperforms other types of refinement. In 3 out of the 10 cases, however, refinement without the use of lemma was able to terminate relatively quickly. We believe that for more complex designs, especially those that are bug-free or with hard-to-find bugs, lemma refinement is essential.

Table 7.9 Verification Condition Nodes and Bits Stats

Test	Concrete Bits	Abstract Nodes	Bits-to-Nodes	Runtime (sec.)
Sorter, W=8	127	26	5.08	0.05
Sorter, W=16	249	26	9.96	0.05
Sorter, W=32	473	26	18.9	0.05
Sorter, W=64	921	26	36.8	0.05
ICRAM	287	90	3.12	0.03
OMU, K=16	1346	344	3.91	0.05
OMU, K=32	3154	1192	2.65	0.1
OMU, K=64	8306	4524	1.88	0.05
OMU, K=128	$2.5x10^4$	$1.7x10^4$	1.47	1.1
DLX, D1	$2.2x10^4$	3945	5.58	0.6
DLX, D2	3552	522	6.8	0.1
DLX, D3	$2.2x10^4$	3915	5.62	1.1
RISC16F84, R1	7286	2904	2.54	148
RISC16F84, R2	7376	2928	2.52	40
RISC16F84, R3	7224	2849	2.54	2.6
X86, X1	$1.5x10^5$	$7x10^4$	2.19	36.5
X86, X2	$1.5x10^5$	$6.7x10^4$	2.28	30.5
X86, X3	2764	3945	1.04	2.0
X86, X4	$1.5x10^5$	$6.7x10^4$	2.28	23

7.8.3 Overall Scalability

Table 7.9 characterizes the sizes of the verification conditions, before and after abstraction, and presents Reveal's runtime on each[4]. The column labeled 'Concrete Bits' shows the number of bits in $conc(X)$, while the column labeled 'Abstract Nodes' shows the number of nodes in $abst(\hat{X})$. The last column shows the ratio between the two, which indicates the average width of the datapath in the design and specifications.

As described earlier, Reveal is able to terminate on all these versions in less than 200 seconds. We noted that there is no particular correlation between the width of the datapath and the verification time. This is attributed to the brute-force datapath abstraction in Reveal. Furthermore, the refinement back-end is robust such that it allows Reveal to terminate despite the control/datapath intermixture.

[4]We show the smaller runtime of ELM and CLM.

Automating both abstraction and refinement plays a significant role in improving the scalability and making Reveal more practical than other approaches. Additionally, aggregating concise lemmas in a persistent database enables efficient incremental verification. Finally, the efficiency of the validity checking and refinement stages is attributed to SMT-based satisfiability checking. The scalability of Reveal can be further improved by performing unbounded model checking based on finite inductive formulations.

7.8.4 Discovering Design and Specification Bugs

Since Reveal produces a counterexample trace, it is as illustrative as 'simulation' in showing the unintended behavior. Furthermore, Reveal proved to be effective in discovering sophisticated bugs that would not have been otherwise discovered. Examples for genuine bugs include the RISC16F84 and MIPS bugs. Finally, Reveal was able to discover other types of unintended behavior, including problems in the specifications, as well as non-determinism in the design that propagated to the outputs. Examples of the earlier include the R2 and X2 variations, and examples of the latter include the problem in the register file in *MIPSBubble*.

It is possible to further improve the output of Reveal by producing additional information alongside the bug trace. Examples of that include actual candidates of the bugs in the design or specification. The use of formal techniques makes it possible to seek such approaches, although their applicability is yet to be determined.

Chapter 8
Conclusions and Future Work

Formal verification of complex hardware systems like microprocessors and microcontrollers has been researched for about two decades. Faced with the state explosion problem, researchers resorted to applying various types of abstraction, in order to filter out design behavior that is orthogonal to the property being verified, and in turn reduce the size of the resulting model. In most of these efforts, however, manual reasoning about designs, properties, and abstractions, has been a major hurdle that slowed down the process overall, and hindered the scalability of formal verification for these types of designs.

Our thesis presents an abstraction-based turn-key verification process for control logic in hardware designs. Scalability is achieved with the use of an automatic counterexample-guided abstraction refinement of the datapath, and automated proofs of safety properties in general, and equivalence in particular. This approach is particularly suited for the verification of designs with wide datapaths and complex control logic. Datapath abstraction allows the approach to focus on the control interactions making it possible to scale up to much larger designs than is possible if verification is carried out at the bit level. Additionally, The scheme's demand-based lemma generation capability eliminates one of the obstacles that had complicated the deployment of formal equivalence tools in the past.

From a practical perspective, hands-free operation and support of Verilog allow the system to be directly used by designers. Reveal, an implementation of the approach, has been tested by university students taking a computer architecture class. Using Reveal as an automatic testing tool prior to synthesis, students were able to hunt for bugs in their

RTL designs, as well as raise their confidence about the correctness of the final design with respect to a certain correctness criterion. During this process, numerous cases were discovered in which unintended behavior was present in the specification and/or implementation of the design, and was fixed using counterexample traces that were generated automatically as well. Three of these cases were design problems in a years-old Verilog code; of those, one was a serious bug that is based on a perfectly plausible scenario. None of these problems were discovered by a comprehensive set of tests that has been used by the class instructors, and is unlikely to have been discovered by traditional simulation approaches.

The capabilities of the approach and practicality of Reveal were further demonstrated by efficiently discovering bugs, or proving the lack thereof, in six Verilog examples that are, amongst publicly available designs, the closest to real-life designs both in terms of size and complexity.

Since Verilog and other design languages were particularly designed to serve as simulation platforms, there has been no clear separation between datapath and control logic that can be represented with a well-defined partition. In general, our approach is resilient against the datapath/control interactions that arise from such ambiguity and lead to false negatives. However, the iterative CEGAR process can still benefit from design methodologies that minimize datapath/control interactions, since those would be, by construction, geared towards datapath abstraction.

We plan to continue developing automated methods that bridge between existing design methodologies and reasoning engines, in order to leverage the latter and maximize its potential, as well as shrink the verification gap. One possible improvement is to automate module-level abstraction, such that entire blocks of hardware implementing self-contained sub-components are abstracted away from the implementation and specification. The dual approach, mostly useful for hunting bugs in medium-size designs, is to perform incremental on-demand abstraction based on performance monitors.

Reveal's usability can be potentially improved with two techniques. Firstly, integrating

unbounded model checking eliminates the need for supplying an unrolling bound, and provides a more complete coverage of correctness for the design. Secondly, debugging can be facilitated to assist in faster identification of design bugs. In particular, cumbersome counterexample traces that stretch over many clock cycles can be replaced with automatically localized 'suggestions' of fixes, which potentially includes the sought design bug. Finally, similar equivalence algorithms can be developed for higher-levels of abstraction such as C [35].

Overall, we believe that formal verification in both hardware and software follows the simple relation: Automation $\wedge$ Efficiency $\rightarrow$ Scalability. We hope that scalability will continue to rise and formal verification methods, like ours, will ultimately be adopted in industrial settings.

Appendices

Appendix A
VERSA Verilog

VERSA, also VErilog Restricted Subset for Abstraction, is designed to combine three main features for the purpose of abstraction-based verification:

- the uniformity of a structural description,
- the word-level granularity of a high-level description,
- limited behavioral modeling that is widely used by designers worldwide.

This subset is suitable for word-level formal verification of control logic described in this thesis, and it allows word-level functionality and datapath abstractions to be more easily inferred. We believe that defining this subset leverages the benefit of abstraction methods by allowing all design descriptions, regardless of their source format (Verilog 2000, System Verilog, VHDL, and other HDLs), to be converted to VERSA and to utilize similar abstraction techniques. In what follows, we describe VERSA and the rationale behind each set of Verilog (un)supported structures.

A.1 Verilog 95

The underlying syntax of VERSA is Verilog 95 [54], which is supported by most Verilog tools including Icarus Verilog [68], a popular and publicly available Verilog compiler. Conceptually, any design can be represented in Verilog 95 regardless of its original HDL source, modulo issues that are orthogonal to Reveal's abstraction-based verification, such as readability of the source code, and scalability for simulation and synthesis.

A.2 Synthesizable Subset

Since Verilog was originally designed as a simulation language, it includes a number of constructs with the sole purpose of facilitating simulation, and bearing no effect on the synthesis process. Since in abstraction-based verification we are interested in modeling the actual hardware that will be ultimately implemented on the chip, VERSA is confined to the synthesizable subset of Verilog. The following Verilog types and constructs are not synthesizable by most tools including Icarus Verilog, and are thus excluded from VERSA:

- Real Constants [54, chap. 2.5.2, p. 8]
- Variables of types real, realtime, and time [54, chap. 3.9, p. 23]
- Strings [54, chap. 6, p. 60]
- Procedural continuous assignments [54, chap. 9.3, p. 104]
- The delay and wait procedural timing controls [54, chap. 9.7, p. 114]
- The event procedural timing control with expressions other than identifiers and posedge/negedge [54, chap. 9.7, p. 114]
- System tasks and functions [54, chap. 14, p. 172]

A.3 Clocking

In VERSA we require that the design has only one clock input (the main clock). Multiple clocks can be derived from the main clock and used to synchronize the logic.

A.4 Explicit Description

Verilog includes a number of features that allow flexible and easy-to-read RTL coding of hardware components. Excluding these features does not compromise the expressiveness of Verilog. Rather, it allows a simplier and more regular representation. The following features are, thus, excluded from VERSA:

- Tasks [54, chap. 10, p. 125]
- Parameters [54, chap. 3.10, p.25]
- Compiler directives [54, chap. 16, p.219], except ‘include and ‘define

A.5 Structural Description

Some additional Verilog features allow the coder to use behavioral description that resembles sequential software. Static preprocessing can remove these features and allow a more ‘regular’ form of the code.

The following constructs are thus excluded from VERSA:

- Parallel blocks [54, chap. 9.8.2, p. 121]
- Named blocks and tasks [54, chap. 11, p. 132]
- Looping [54, chap. 9.6, p. 111]

A.6 Abstraction-Oriented Description

According to [54, chap. 6, p. 50] LHS of continuous assignments can include bit- and part-selects with constant indexing. VERSA requires constant indexing in *all* expressions involving extraction from a 1-dimensional bit-vectors. In particular, (1) procedural assignments to bit-vectors with variable indexing in the LHS should be replaced with multiple assignments representing each bit- or part-select; (2) expressions appearing in RHS of assignments with variable indexing applied on bit-vectors should be replaced with ?: expressions that convert them to constant indexing. VERSA also restricts the use of non-constant repeat count [54, chap. 4.1, p. 27] and shifting by a non-constant value.

Since the approach is premised on RTL Verilog, VERSA excludes gate- and switch-level modeling [54, chap. 7, p. 55] and user defined premitives [54, chap. 8, p. 87]. Gates should be replaced with continuous assignments. VERSA also requires that wire is the only

net type. Other net types such as buses, pullups, pulldowns, and supplies [54, chap. 3.7, p. 17], should be eliminated.

The following constructs are allowed in a VERSA description, but are ignored since they bear no effect on the verification on Reveal:

- Vectored and scalared variables [54, chap. 3.3.2, p. 15]
- Minimum, typical, and maximum delay expressions [54, chap. 4.3, p. 42]
- Strengths [54, chap. 6.1.4, p. 53]
- Procedural timing controls [54, chap. 9.7, p. 114]
- Specify-blocks [54, chap. 13, p. 152]

A.7 Memories

VERSA allows the modeling of memories with 2-dimensional bit-vectors. LHS and RHS expressions that involve extraction (with dynamic indexing) are treated as write and read ports, respectively. No extraction with constant indexing is allowed with memory 2-dimensional variables.

Appendix B
Configuration Directives

Reveal's specific choice for abstraction, refinement and solving are controlled via a set of configuration directives. We divide them into three groups following Section 6.2.1.

B.1 Algorithmic Behavior

The following arguments control the abstraction, refinement, and solving steps:

- **alg_type [optional, default: abst_ref]**. Through this argument the user can control the type of verification algorithm. 'abst_ref' indicates the use of DP-CEGAR, while 'bit_blast' indicates the use of YICES' SMT(BV) by representing $conc(X)$ with the native bit-vector representation in YICES.
- **conc_min_type [optional, default: all_muses]**. Counterexample minimization using MUSes can be controlled using this argument. The values 'all_muses' and 'one_muses' allows the use of all or one MUS in each refinement iteration, while the argument 'none' turns off MUS-based minimization.
- **abst_min_type [optional, default: none]**. Counterexample minimization at the abstract level can be done through this option.
- **lemma_db [optional]**. This argument activates the lemma database and specifies the name of the file used to store the lemmas.
- **sim_simplifications [optional, default: 1]**. When disabled with value '0', simulation simplifications are not used (see Section 6.2.2).

- **max_iter [optional, default: 0]**. When a non-zero value is used, refinement iterations are limited to the given number.
- **abst_solver [optional, default: yices_api]**. This specifies the type of the abstract solver used, which is one of 'yices_api', 'yices', 'stp', 'ario', and 'bat'.
- **abst_logic [optional, default: euf]**. This specifies the abstract logic used, which is one of 'euf' or 'clu'.
- **camus_timeout [optional, default: 5]**. This specifies the number of seconds after which CAMUS times-out during counterexample minimization.
- **camus_groups [optional, default: 0]**. A non-zero value directs Reveal to group the constraints in the violation into the given number of groups (arbitrarily) before passing to CAMUS. Constraints that are grouped together in CAMUS will be enabled (included in the MUS) or disabled (excluded from the MUS) together.
- **camus_group_sizes [optional, default: 0]**. A non-zero value directs Reveal to group the constraints in the violation such that each group includes the specified number of constraints. See the 'camus_groups' argument above.
- **camus_max_muses [optional, default: 0]**. A non-zero value directs CAMUS to limit the number of MUSes used for refinement to the given number.

B.2 Input Specifications

The following arguments control the way Reveal models the input design, as well as necessary information to launch the verification.

B.2.1 Design Modeling

- **clock_sig [optional]**. This specifies the name of the clock signal. This is the main clock input driving the design. Multiple clocks can be derived from the main clock, and they can all (including the main clock) be used to synchronize the design's logic

on either edge or value. Purely combinational designs can also be specified, in which case no clock signal is needed.

- **clock_model [optional, default: init_0_oscillating]**. This argument tells the simulator the way the main clock signal is modeled. Reveal's supports the 'init_0_oscillating' mode, where the clock is automatically initialized with 0 in cycle 0, and oscillates (hi to lo or vice versa) in each new cycle; and the 'init_0_posedge', which forces the clock to be 0 at cycle 0 but to have a single positive edge in each cycle. The latter is allowed for designs that exclude negative clock edge synchronization, and prevents spending two cycles for a single clock period.
- **prop_cycle [mandatory]**. This specifies the number of cycles the design has to be unfolded to generate $conc(X)$.
- **truncate_rhs [optional, default: 0]**. When turned on, Reveal will truncate RHS expressions that are assigned to a wider LHS signals to enforce size compatibility. By default, this is disabled and such design input produces an error.
- **extend_rhs [optional, default: 0]**. When turned on, Reveal will 0-extend RHS expressions that are assigned to a narrower LHS signals to enforce size compatibility. By default, this is disabled and such design input produces an error.
- **mem_map [optional]**. The given file specifies a mapping for memory arrays. This is useful for enforcing initialization for memories. Each line in this file specifies the name of the memory array, and the new name of its corresponding memory array used in cycle 0.

B.2.2 Design Information

- **design_file [mandatory]**. Specifies the name of the Verilog file including the top module.
- **design_type [optional, default: verilog]**. Specifies the format type of the input. If given 'hr', Reveal will attempt to read a binary HR representation of the transition

function.

- **prop_file [mandatory]**. Specifies the name of the file including the property.
- **prop_type [optional, default: verilog]**. Specifies the format type of the property file similarly to 'design_type'.
- **prop_sig [mandatory]**. This specifies the name of the signal represented the property defined in 'prop_file'.
- **top_module [optional]**. This specifies the name of the top module for multi-module designs.

B.3 Output Specifications

The arguments described here control Reveal's output on the screen and as numerous files. We divide these into arguments that control the back-end of Reveal for the purpose of resuming the verification with other tools, and arguments for the sole purpose of debugging and understanding the operation of Reveal.

B.3.1 Back-end

- **dump_{init|final}_formula_in_{verilog|uclid} [optional]**. When activated with value '1', Reveal dumps the initial ($conc(X)$) or final formula ($\hat{\varphi}^N(\hat{X}, p)$, where N is the number of refinement iterations) in Verilog or UCLID.
- **trace_signals [optional]**. This specifies to Reveal which signals should be included for viewing in the GUI back-end.

B.3.2 Debugging

- **dump_design_modeling [optional]**. The transition relation of the design is textually dumped to the file specified in this option.

- **dump_design_hr [optional, default: 0]**. This specifies whether to dump the transition relation of the design in the 'HR' format.
- **dump_model [optional, default: 0]**. The transition relation of the design is textually dumped to the screen.
- **dump_cex [optional, default: 0]**. All abstract counterexamples are dumped (to stdout) when this option is turned on.
- **dump_viol [optional, default: 0]**. The violation computed in each iteration is textually dumped to the screen.
- **dump_ref [optional, default: 0]**. The refinement computed in each iteration is textually dumped to the screen.
- **dump_stats [optional]**. Dumps verification statistics to the specified file.
- **sim_signals [optional]**. A list of signals to be printed during simulation.
- **verbosity [optional, default: 1]**. Verbosity of Reveal's output ranges from level 0 to 3, where level 1 shows the CEGAR loop and the time spent on the abstraction versus refinement iterations.

Appendix C

MIPS Bubble-to-Bubble Counterexample Trace

```
1.  Reveal - University of Michigan, Ann Arbor
2.  Version:1.1
3.
4.  CONFIGURATION:
5.  ************************************************
6.  abst_logic = euf
7.  abst_min_type = none
8.  abst_solver = yices_api
9.  alg_type = abst_ref
10. aux_file =
11. bound_on_abst_vars = 0
12. camus_group_size = 0
13. camus_groups = 0
14. camus_max_muses = 100
15. camus_timeout = 5
16. clock_model = init_0_oscillating
17. clock_sig = clock
18. conc_min_type = all_muses
19. design_file = wenisch_identical.v
20. design_type = verilog
21. dump_cex =
22. dump_design_hr =
23. dump_design_modeling =
24. dump_final_formula_in_uclid =
25. dump_final_formula_in_verilog =
26. dump_init_formula_in_uclid =
27. dump_init_formula_in_verilog =
28. dump_model = 470_identical.model
29. dump_ref =
30. dump_stats =
31. dump_viol =
32. experiment = 0
```

33. experiment_coi = 0
34. extend_rhs = 0
35. interactive_debugger = 0
36. lemma_db =
37. max_iter = 0
38. mem_map = wenisch_identical.mem_map
39. prop_cycle = 13
40. prop_file = prop.v
41. prop_sig = prop
42. prop_type = verilog
43. sim_signals =
44. sim_simplifications = on
45. top_module = pipeline
46. trace_signals =
47. truncate_rhs = 0
48. variables_abstraction = ints_interpreted_consts
49. verbosity = 0
50. **
51. Reveal started...
52. -I- model dumped to: 470_identical.model
53. -I- ********************
54. -I- Property is Violated!
55. -I- ********************
56. -I-
57. Counterexample Trace:
58. ====================
59. Cycle 0
60. ——————
61. mem_wb_NPC_design1
62. = 0000000000000000000000000000000000000000000000000000000000000001
63. **pipeline_design1_0$mem_wb_dest_reg_idx**
64. **= 00000**
65. pipeline_design1_0$mem_wb_take_branch
66. = 1
67. **pipeline_design1_0$wb_reg_wr_data_out**
68. **= 0000000000000000000000000000000000000000000000000000000000000001**
69. **pipeline_design1_0$wb_reg_wr_en_out**
70. **= 1**
71. **pipeline_design1_0$wb_reg_wr_idx_out**
72. **= 00000**
73. pipeline_design1_0$wb_stage_design1_0$result_mux
74. = 0000000000000000000000000000000000000000000000000000000000000001
75. pipeline_design2_0$mem_wb_dest_reg_idx
76. = 00001
77. **pipeline_design2_0$wb_reg_wr_idx_out**

78. **= 00001**
79.
80. Cycle 1
81. ——————
82. id_ex_IR_design1
83. = 01000111111111110000010000011111
84. id_ex_IR_design2
85. = 01000111111111110000010000011111
86. if_id_IR_design1
87. = 01000111111111110000010000011111
88. if_id_IR_design2
89. = 01000111111111110000010000011111
90. pipeline_design1_0$ex_mem_alu_result
91. = 0000000000000000000000000000000000000000000000000000000000000000
92. pipeline_design1_0$ex_mem_dest_reg_idx
93. = 11111
94. pipeline_design1_0$id_ex_rega
95. = 0000000000000000000000000000000000000000000000000000000000000000
96. pipeline_design1_0$if_stage_design1_0$PC_reg
97. = 0000000000000000000000000000000000000000000000000000000000000000
98. pipeline_design1_0$mem_wb_dest_reg_idx
99. = 11111
100. pipeline_design1_0$mem_wb_result
101. = 0000000000000000000000000000000000000000000000000000000000000000
102. pipeline_design2_0$ex_mem_alu_result
103. = 0000000000000000000000000000000000000000000000000000000000000000
104. pipeline_design2_0$ex_mem_dest_reg_idx
105. = 11111
106. pipeline_design2_0$id_ex_rega
107. = 0000000000000000000000000000000000000000000000000000000000000000
108. pipeline_design2_0$if_stage_design2_0$PC_reg
109. = 0000000000000000000000000000000000000000000000000000000000000000
110. pipeline_design2_0$mem_wb_dest_reg_idx
111. = 11111
112. pipeline_design2_0$mem_wb_result
113. = 0000000000000000000000000000000000000000000000000000000000000000
114.
115. Cycle 2
116. ——————
117. if_IR_out_design1
118. = 11100000000111111111111111111111
119. if_IR_out_design2
120. = 11100000000111111111111111111111
121. if_NPC_out_design1
122. = 0000000000000000000000000000000000000000000000000000000000000100

123. if_NPC_out_design2
124. = 0000000000000000000000000000000000000000000000000000000000000100
125. if_id_IR_design1
126. = 01000111111111110000010000011111
127. if_id_IR_design2
128. = 01000111111111110000010000011111
129. mem2proc_data_design1
130. = 11100000000111111111111111111111
131. mem2proc_data_design2
132. = 11100000000111111111111111111111
133. pipeline_design1_0$ex_mem_alu_result
134. = 0000000000000000000000000000000000000000000000000000000000000000
135. pipeline_design1_0$ex_mem_dest_reg_idx
136. = 11111
137. pipeline_design1_0$id_cond_branch_out
138. = 0
139. pipeline_design1_0$id_ex_rega
140. = 0000000000000000000000000000000000000000000000000000000000000000
141. pipeline_design1_0$id_uncond_branch_out
142. = 0
143. pipeline_design1_0$if_stage_design1_0$PC_plus_4
144. = 0000000000000000000000000000000000000000000000000000000000000100
145. pipeline_design1_0$if_stage_design1_0$PC_reg
146. = 0000000000000000000000000000000000000000000000000000000000000000
147. pipeline_design1_0$if_stage_design1_0$next_PC
148. = 0000000000000000000000000000000000000000000000000000000000000100
149. pipeline_design1_0$mem_result_out
150. = 0000000000000000000000000000000000000000000000000000000000000000
151. pipeline_design1_0$mem_wb_dest_reg_idx
152. = 11111
153. pipeline_design1_0$mem_wb_result
154. = 0000000000000000000000000000000000000000000000000000000000000000
155. pipeline_design1_0$proc2Imem_addr
156. = 0000000000000000000000000000000000000000000000000000000000000000
157. pipeline_design1_0$wb_reg_wr_data_out
158. = 0000000000000000000000000000000000000000000000000000000000000000
159. pipeline_design1_0$wb_reg_wr_idx_out
160. = 11111
161. pipeline_design1_0$wb_stage_design1_0$result_mux
162. = 0000000000000000000000000000000000000000000000000000000000000000
163. pipeline_design2_0$ex_mem_alu_result
164. = 0000000000000000000000000000000000000000000000000000000000000000
165. pipeline_design2_0$ex_mem_dest_reg_idx
166. = 11111
167. pipeline_design2_0$id_cond_branch_out

168. = 0
169. pipeline_design2_0$id_ex_rega
170. = 0000000000000000000000000000000000000000000000000000000000000000
171. pipeline_design2_0$id_uncond_branch_out
172. = 0
173. pipeline_design2_0$if_stage_design2_0$PC_plus_4
174. = 0000000000000000000000000000000000000000000000000000000000000100
175. pipeline_design2_0$if_stage_design2_0$PC_reg
176. = 0000000000000000000000000000000000000000000000000000000000000000
177. pipeline_design2_0$if_stage_design2_0$next_PC
178. = 0000000000000000000000000000000000000000000000000000000000000100
179. pipeline_design2_0$mem_result_out
180. = 0000000000000000000000000000000000000000000000000000000000000000
181. pipeline_design2_0$mem_wb_dest_reg_idx
182. = 11111
183. pipeline_design2_0$mem_wb_result
184. = 0000000000000000000000000000000000000000000000000000000000000000
185. pipeline_design2_0$proc2Imem_addr
186. = 0000000000000000000000000000000000000000000000000000000000000000
187. pipeline_design2_0$wb_reg_wr_data_out
188. = 0000000000000000000000000000000000000000000000000000000000000000
189. pipeline_design2_0$wb_reg_wr_idx_out
190. = 11111
191. pipeline_design2_0$wb_stage_design2_0$result_mux
192. = 0000000000000000000000000000000000000000000000000000000000000000
193. proc2mem_addr_design1
194. = 0000000000000000000000000000000000000000000000000000000000000000
195. proc2mem_addr_design2
196. = 0000000000000000000000000000000000000000000000000000000000000000
197.
198. Cycle 3
199. ——————
200. id_ex_IR_design1
201. = 01000111111111110000010000011111
202. id_ex_IR_design2
203. = 01000111111111110000010000011111
204. **if_id_IR_design1**
205. **= 11100000000111111111111111111111**
206. if_id_IR_design2
207. = 11100000000111111111111111111111
208. if_id_NPC_design1
209. = 0000000000000000000000000000000000000000000000000000000000000100
210. if_id_NPC_design2
211. = 0000000000000000000000000000000000000000000000000000000000000100
212. pipeline_design1_0$id_ex_cond_branch

213. = 0
214. pipeline_design1_0$id_ex_uncond_branch
215. = 0
216. pipeline_design1_0$if_stage_design1_0$PC_reg
217. = 0000000000000000000000000000000000000000000000000000000000000100
218. pipeline_design1_0$mem_wb_dest_reg_idx
219. = 11111
220. pipeline_design1_0$mem_wb_result
221. = 0000000000000000000000000000000000000000000000000000000000000000
222. pipeline_design2_0$id_ex_cond_branch
223. = 0
224. pipeline_design2_0$id_ex_uncond_branch
225. = 0
226. pipeline_design2_0$if_stage_design2_0$PC_reg
227. = 0000000000000000000000000000000000000000000000000000000000000100
228. pipeline_design2_0$mem_wb_dest_reg_idx
229. = 11111
230. pipeline_design2_0$mem_wb_result
231. = 0000000000000000000000000000000000000000000000000000000000000000
232.
233. Cycle 4
234. ——————
235. if_id_IR_design1
236. = 11100000000111111111111111111111
237. if_id_IR_design2
238. = 11100000000111111111111111111111
239. if_id_NPC_design1
240. = 0000000000000000000000000000000000000000000000000000000000000100
241. if_id_NPC_design2
242. = 0000000000000000000000000000000000000000000000000000000000000100
243. pipeline_design1_0$ex_take_branch_out
244. = 0
245. pipeline_design1_0$id_ex_cond_branch
246. = 0
247. pipeline_design1_0$id_ex_uncond_branch
248. = 0
249. pipeline_design1_0$id_rega_out
250. = 0000000000000000000000000000000000000000000000000000000000000001
251. pipeline_design1_0$id_stage_design1_0$decoder_design1_0$op_code
252. = 111000
253. pipeline_design1_0$id_stage_design1_0$decoder_design1_0$op_code1
254. = 111000
255. pipeline_design1_0$id_stage_design1_0$ra_idx
256. = 00000
257. pipeline_design1_0$id_stage_design1_0$regf_design1_0$rda_reg

258. = 0000000000000000000000000000000000000000000000000000000000000001
259. pipeline_design1_0$if_stage_design1_0$PC_reg
260. = 0000000000000000000000000000000000000000000000000000000000000100
261. pipeline_design1_0$mem_wb_dest_reg_idx
262. = 11111
263. pipeline_design1_0$mem_wb_result
264. = 0000000000000000000000000000000000000000000000000000000000000000
265. pipeline_design1_0$wb_reg_wr_data_out
266. = 0000000000000000000000000000000000000000000000000000000000000000
267. pipeline_design1_0$wb_reg_wr_en_out
268. = 0
269. pipeline_design1_0$wb_reg_wr_idx_out
270. = 11111
271. pipeline_design1_0$wb_stage_design1_0$result_mux
272. = 0000000000000000000000000000000000000000000000000000000000000000
273. pipeline_design2_0$ex_take_branch_out
274. = 0
275. pipeline_design2_0$id_alu_func_out
276. = 00000
277. pipeline_design2_0$id_ex_cond_branch
278. = 0
279. pipeline_design2_0$id_ex_uncond_branch
280. = 0
281. pipeline_design2_0$id_opa_select_out
282. = 10
283. pipeline_design2_0$id_opb_select_out
284. = 10
285. pipeline_design2_0$id_rega_out
286. = 0000000000000000000000000000000000000000000000000000000000000000
287. pipeline_design2_0$id_stage_design2_0$decoder_design2_0$op_code
288. = 111000
289. pipeline_design2_0$id_stage_design2_0$decoder_design2_0$op_code1
290. = 111000
291. pipeline_design2_0$id_stage_design2_0$ra_idx
292. = 00000
293. pipeline_design2_0$id_stage_design2_0$regf_design2_0$rda_reg
294. = 0000000000000000000000000000000000000000000000000000000000000000
295. pipeline_design2_0$if_stage_design2_0$PC_reg
296. = 0000000000000000000000000000000000000000000000000000000000000100
297. pipeline_design2_0$mem_wb_dest_reg_idx
298. = 11111
299. pipeline_design2_0$mem_wb_result
300. = 0000000000000000000000000000000000000000000000000000000000000000
301. pipeline_design2_0$wb_reg_wr_data_out
302. = 0000000000000000000000000000000000000000000000000000000000000000

303. pipeline_design2_0$wb_reg_wr_en_out
304. = 0
305. pipeline_design2_0$wb_reg_wr_idx_out
306. = 11111
307. pipeline_design2_0$wb_stage_design2_0$result_mux
308. = 0000000000000000000000000000000000000000000000000000000000000000
309.
310. Cycle 5
311. —————
312. **id_ex_IR_design1**
313. **= 11100000000111111111111111111111**
314. **id_ex_IR_design2**
315. **= 11100000000111111111111111111111**
316. **id_ex_NPC_design1**
317. **= 0000000000000000000000000000000000000000000000000000000000000100**
318. **id_ex_NPC_design2**
319. **= 0000000000000000000000000000000000000000000000000000000000000100**
320. pipeline_design1_0$ex_mem_take_branch
321. = 0
322. **pipeline_design1_0$id_ex_rega**
323. **= 0000000000000000000000000000000000000000000000000000000000000001**
324. pipeline_design1_0$if_stage_design1_0$PC_reg
325. = 0000000000000000000000000000000000000000000000000000000000000100
326. pipeline_design2_0$ex_mem_take_branch
327. = 0
328. pipeline_design2_0$id_ex_alu_func
329. = 00000
330. pipeline_design2_0$id_ex_opa_select
331. = 10
332. pipeline_design2_0$id_ex_opb_select
333. = 10
334. pipeline_design2_0$id_ex_rega
335. = 0000000000000000000000000000000000000000000000000000000000000000
336. pipeline_design2_0$if_stage_design2_0$PC_reg
337. = 0000000000000000000000000000000000000000000000000000000000000100
338.
339. Cycle 6
340. —————
341. id_ex_IR_design1
342. = 000xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
343. id_ex_IR_design2
344. = 000xxxxx111111111111111111111111
345. id_ex_NPC_design1
346. = 0000000000000000000000000000000000000000000000000000000000000100
347. id_ex_NPC_design2

348. = 0000000000000000000000000000000000000000000000000000000000000100
349. pipeline_design1_0$ex_mem_take_branch
350. = 0
351. pipeline_design1_0$ex_stage_design1_0$id_ex_IR_aux
352. = 000
353. pipeline_design1_0$id_cond_branch_out
354. = 0
355. pipeline_design1_0$id_ex_rega
356. = 0000000000000000000000000000000000000000000000000000000000000001
357. pipeline_design1_0$id_uncond_branch_out
358. = 0
359. pipeline_design1_0$if_stage_design1_0$PC_reg
360. = 0000000000000000000000000000000000000000000000000000000000000100
361. pipeline_design2_0$ex_alu_result_out
362. = 0000000000000000000000000000000000000000000000000000000000000000
363. pipeline_design2_0$ex_mem_take_branch
364. = 0
365. pipeline_design2_0$ex_stage_design2_0$br_disp
366. = 1111111111111111111111111111111111111111111111111111111111111100
367. pipeline_design2_0$ex_stage_design2_0$id_ex_IR_aux
368. = 000
369. pipeline_design2_0$ex_stage_design2_0$opa_mux_out
370. = 0000000000000000000000000000000000000000000000000000000000000100
371. pipeline_design2_0$ex_stage_design2_0$opb_mux_out
372. = 1111111111111111111111111111111111111111111111111111111111111100
373. pipeline_design2_0$id_cond_branch_out
374. = 0
375. pipeline_design2_0$id_ex_alu_func
376. = 00000
377. pipeline_design2_0$id_ex_opa_select
378. = 10
379. pipeline_design2_0$id_ex_opb_select
380. = 10
381. pipeline_design2_0$id_ex_rega
382. = 0000000000000000000000000000000000000000000000000000000000000000
383. pipeline_design2_0$id_uncond_branch_out
384. = 0
385. pipeline_design2_0$if_stage_design2_0$PC_reg
386. = 0000000000000000000000000000000000000000000000000000000000000100
387.
388. Cycle 7
389. ——————
390. pipeline_design1_0$id_ex_cond_branch
391. = 0
392. pipeline_design1_0$id_ex_uncond_branch

393. = 0
394. pipeline_design1_0$if_stage_design1_0$PC_reg
395. = 0000000000000000000000000000000000000000000000000000000000000100
396. pipeline_design2_0$ex_mem_alu_result
397. = 0000000000000000000000000000000000000000000000000000000000000000
398. pipeline_design2_0$id_ex_cond_branch
399. = 0
400. pipeline_design2_0$id_ex_uncond_branch
401. = 0
402. pipeline_design2_0$if_stage_design2_0$PC_reg
403. = 0000000000000000000000000000000000000000000000000000000000000100
404.
405. Cycle 8
406. ——————
407. pipeline_design1_0$ex_take_branch_out
408. = 0
409. pipeline_design1_0$id_ex_cond_branch
410. = 0
411. pipeline_design1_0$id_ex_uncond_branch
412. = 0
413. pipeline_design1_0$if_stage_design1_0$PC_reg
414. = 0000000000000000000000000000000000000000000000000000000000000100
415. pipeline_design2_0$ex_mem_alu_result
416. = 0000000000000000000000000000000000000000000000000000000000000000
417. pipeline_design2_0$ex_take_branch_out
418. = 0
419. pipeline_design2_0$id_ex_cond_branch
420. = 0
421. pipeline_design2_0$id_ex_uncond_branch
422. = 0
423. pipeline_design2_0$if_stage_design2_0$PC_reg
424. = 0000000000000000000000000000000000000000000000000000000000000100
425. pipeline_design2_0$if_stage_design2_0$next_PC
426. = 0000000000000000000000000000000000000000000000000000000000000000
427.
428. Cycle 9
429. ——————
430. pipeline_design1_0$ex_mem_take_branch
431. = 0
432. pipeline_design1_0$if_stage_design1_0$PC_reg
433. = 0000000000000000000000000000000000000000000000000000000000000100
434. pipeline_design2_0$ex_mem_take_branch = 0
435. pipeline_design2_0$if_stage_design2_0$PC_reg
436. = 0000000000000000000000000000000000000000000000000000000000000000
437.

438. Cycle 10
439. ————
440. pc_equiv_curr_cycle
441. = 0
442. pipeline_design1_0$ex_mem_take_branch
443. = 0
444. pipeline_design1_0$if_stage_design1_0$PC_reg
445. = 0000000000000000000000000000000000000000000000000000000000000100
446. pipeline_design2_0$ex_mem_take_branch
447. = 0
448. pipeline_design2_0$if_stage_design2_0$PC_reg
449. = 0000000000000000000000000000000000000000000000000000000000000000
450.
451. Cycle 11
452. ————
453. pc_equiv_always
454. = 0
455. pipeline_design1_0$if_stage_design1_0$PC_reg
456. = 0000000000000000000000000000000000000000000000000000000000000100
457. pipeline_design2_0$if_stage_design2_0$PC_reg
458. = 0000000000000000000000000000000000000000000000000000000000000000
459.
460. Cycle 12
461. ————
462. pc_equiv_always
463. = 0
464. **pipeline_design1_0$if_stage_design1_0$PC_reg**
465. **= 0000000000000000000000000000000000000000000000000000000000000100**
466. **pipeline_design2_0$if_stage_design2_0$PC_reg**
467. **= 0000000000000000000000000000000000000000000000000000000000000000**
468.
469. -I- Total iterations: 1
470.
471. Reveal finished.
472.
473. Abst Solving Time: 0.08
474. Total Time: 0.26

Appendix D

MIPS Spec-to-Pipe Counterexample Trace

```
1. Reveal - University of Michigan, Ann Arbor
2. Version:1.1
3.
4. CONFIGURATION:
5. ***********************************************
6. abst_logic = euf
7. abst_min_type = none
8. abst_solver = yices_api
9. alg_type = abst_ref
10. aux_file =
11. bound_on_abst_vars = 0
12. camus_group_size = 0
13. camus_groups = 5
14. camus_max_muses = 100
15. camus_timeout = 5
16. clock_model = init_0_oscillating
17. clock_sig = clock
18. conc_min_type = all_muses
19. design_file = alphatest.v
20. design_type = verilog
21. dump_cex =
22. dump_design_hr =
23. dump_design_modeling =
24. dump_final_formula_in_uclid =
25. dump_final_formula_in_verilog =
26. dump_init_formula_in_uclid =
27. dump_init_formula_in_verilog =
28. dump_model = ../work/alphatest.model
29. dump_ref =
30. dump_stats = ../work/alphatest.stats
31. dump_viol =
32. experiment = 0
```

33. experiment_coi = 0
34. extend_rhs = 0
35. interactive_debugger = 0
36. lemma_db = ../work/alphatest.lemmas_db
37. max_iter = 0
38. mem_map = alphatest.mem_map
39. prop_cycle = 12
40. prop_file = prop.v
41. prop_sig = prop
42. prop_type = verilog
43. sim_signals =
44. sim_simplifications = on
45. top_module = alphatest
46. trace_signals =
47. truncate_rhs = 0
48. variables_abstraction = ints_interpreted_consts
49. verbosity = 1
50. **
51.
52. Reveal started...
53.
54. -D- Loading the design...
55. -D- 1-bit registers: 1147
56. -D- Loading the property...
57. -D- Done.
58. -I- model dumped to: ../work/alphatest.model
59. Added clock:INIT[clock] := 1'd0
60. Added clock:NSF[clock] := !clock
61. -D- Simulating..
62. -D- Done.
63. -I-
64. -I- Creating an empty Lemma DB file: ../work/alphatest.lemmas_db
65. -I-
66. -I- ********************
67. -I- Property is Violated!
68. -I- ********************
69. -I-
70. Counterexample Trace:
71. ====================
72.
73. Cycle 0
74. ———————
75. f1
76. = 1
77. f2

78. = 1
79. halt_prev_pipeline
80. = 0
81. pipeline_0if_stage_0PC_reg
82. = 0000000000000000000000000000000000000000000000000000000000110000
83. pipeline_spec_0$if_stage_spec_0$PC_reg
84. = 0000000000000000000000000000000000000000000000000000000000110000
85.
86. Cycle 1
87. ——————
88. f1_acc
89. = 1
90. f2_acc
91. = 1
92. halt_prev_pipeline
93. = 0
94. id_ex_IR_impl
95. = 01000111111111110000010000011111
96. if_id_IR_impl
97. = 01000111111111110000010000011111
98. if_id_NPC_impl
99. = 0000000000000000000000000000000000000000000000000000000000000000
100. if_id_valid_inst_impl
101. = 0
102. mem2proc_fetch_data_spec
103. = 11101011110000000000000000110001
104. mem_wb_IR_spec
105. = 111xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
106. pipeline_impl_0$ex_mem_alu_result
107. = 0000000000000000000000000000000000000000000000000000000000000000
108. pipeline_impl_0$ex_mem_dest_reg_idx
109. = 11111
110. pipeline_impl_0$ex_mem_halt
111. = 0
112. pipeline_impl_0$ex_mem_illegal
113. = 0
114. pipeline_impl_0$ex_mem_rd_mem
115. = 0
116. pipeline_impl_0$ex_mem_take_branch
117. = 0
118. pipeline_impl_0$ex_mem_wr_mem
119. = 0
120. pipeline_impl_0$ex_value_a
121. = 0000000000000000000000000000000000000000000000000000000000000000
122. pipeline_impl_0$id_ex_alu_func

123. = 00000
124. pipeline_impl_0$id_ex_cond_branch
125. = 0
126. pipeline_impl_0$id_ex_dest_reg_idx
127. = 11111
128. pipeline_impl_0$id_ex_fwd_a
129. = 00
130. pipeline_impl_0$id_ex_fwd_b
131. = 00
132. pipeline_impl_0$id_ex_halt
133. = 0
134. pipeline_impl_0$id_ex_illegal
135. = 0
136. pipeline_impl_0$id_ex_opa_select
137. = 00
138. pipeline_impl_0$id_ex_opb_select
139. = 00
140. pipeline_impl_0$id_ex_rd_mem
141. = 0
142. pipeline_impl_0$id_ex_rega
143. = 0000000000000000000000000000000000000000000000000000000000000000
144. pipeline_impl_0$id_ex_regb
145. = 0000000000000000000000000000000000000000000000000000000000000000
146. pipeline_impl_0$id_ex_uncond_branch
147. = 0
148. pipeline_impl_0$id_ex_wr_mem
149. = 0
150. **pipeline_impl_0id_stage_0fwd_check_EX**
151. **= 11110**
152. **pipeline_impl_0id_stage_0fwd_check_MEM**
153. **= 11110**
154. pipeline_impl_0if_stage_0PC_reg
155. = 0000000000000000000000000000000000000000000000000000000000000000
156. pipeline_impl_0$mem_wb_dest_reg_idx
157. = 11111
158. pipeline_impl_0$mem_wb_result
159. = 0000000000000000000000000000000000000000000000000000000000000000
160. pipeline_impl_0$proc2Dmem_command
161. = 00
162. pipeline_spec_0ex_stage_spec0opb_mux_out
163. = 0000000000000000000000000000000000000000000000000000000000000000
164. pipeline_spec_0$id_alu_func_out
165. = 00000
166. pipeline_spec_0$id_opa_select_out
167. = 01

168. pipeline_spec_0$id_opb_select_out
169. = 00
170. pipeline_spec_0$id_regb_out
171. = 0000000000000000000000000000000000000000000000000000000000000000
172. pipeline_spec_0if_stage_spec0PC_reg
173. = 0000000000000000000000000000000000000000000000000000000000000000
174. pipeline_spec_0$proc2Dmem_command
175. = 00
176. pipeline_spec_0$proc2Imem_addr
177. = 0000000000000000000000000000000000000000000000000000000000000000
178. proc2mem_command_impl
179. = 01
180. proc2mem_command_spec
181. = 00
182. proc2mem_fetch_addr_spec
183. = 0000000000000000000000000000000000000000000000000000000000000xxx
184. spec_mem_idx2
185. = 0000000000000000000000000000000000000000000000000000000000000
186.
187. Cycle 2
188. ——————
189. f1
190. = 1
191. f1_acc
192. = 1
193. f2
194. = 1
195. f2_acc
196. = 1
197. halt_prev_pipeline
198. = 0
199. if_IR_out_impl
200. = 11101011110000000000000000110001
201. if_NPC_out_impl
202. = 0000000000000000000000000000000000000000000000000000000000000100
203. if_id_IR_impl
204. = 01000111111111110000010000011111
205. if_id_NPC_impl
206. = 0000000000000000000000000000000000000000000000000000000000000000
207. if_id_valid_inst_impl
208. = 0
209. if_valid_inst_out_impl
210. = 1
211. impl_mem_idx
212. = 0000000000000000000000000000000000000000000000000000000000000

213. mem2proc_data_impl
214. = 11101011110000000000000000110001
215. mem2proc_fetch_data_spec
216. = 11101011110000000000000000110001
217. mem_wb_IR_spec
218. = 11101011110000000000000000110001
219. mem_wb_NPC_spec
220. = 0000000000000000000000000000000000000000000000000000000000000100
221. pipeline_0$id_illegal_out
222. = 0
223. pipeline_0if_stage_0PC_reg
224. = 0000000000000000000000000000000000000000000000000000000000110000
225. pipeline_commit_NPC_spec
226. = 0000000000000000000000000000000000000000000000000000000000000100
227. pipeline_error_status_impl
228. = 0000
229. pipeline_error_status_spec
230. = 0000
231. pipeline_impl_0$ex_alu_result_out
232. = 0000000000000000000000000000000000000000000000000000000000000000
233. pipeline_impl_0$ex_mem_alu_result
234. = 0000000000000000000000000000000000000000000000000000000000000000
235. pipeline_impl_0$ex_mem_dest_reg_idx
236. = 11111
237. pipeline_impl_0$ex_mem_halt
238. = 0
239. pipeline_impl_0$ex_mem_illegal
240. = 0
241. pipeline_impl_0$ex_mem_rd_mem
242. = 0
243. pipeline_impl_0$ex_mem_take_branch
244. = 0
245. pipeline_impl_0$ex_mem_wr_mem
246. = 0
247. pipeline_impl_0ex_stage_0opa_mux_out
248. = 0000000000000000000000000000000000000000000000000000000000000000
249. pipeline_impl_0ex_stage_0opb_mux_out
250. = 0000000000000000000000000000000000000000000000000000000000000000
251. pipeline_impl_0$ex_take_branch_out
252. = 0
253. pipeline_impl_0$ex_value_a
254. = 0000000000000000000000000000000000000000000000000000000000000000
255. pipeline_impl_0$ex_value_b
256. = 0000000000000000000000000000000000000000000000000000000000000000
257. pipeline_impl_0$fwd_a

258. = 00
259. pipeline_impl_0$fwd_b
260. = 00
261. pipeline_impl_0$id_alu_func_out
262. = 00000
263. pipeline_impl_0$id_cond_branch_out
264. = 0
265. pipeline_impl_0$id_dest_reg_idx_out
266. = 11111
267. pipeline_impl_0$id_ex_alu_func
268. = 00000
269. pipeline_impl_0$id_ex_cond_branch
270. = 0
271. pipeline_impl_0$id_ex_dest_reg_idx
272. = 11111
273. pipeline_impl_0$id_ex_fwd_a
274. = 00
275. pipeline_impl_0$id_ex_fwd_b
276. = 00
277. pipeline_impl_0$id_ex_halt
278. = 0
279. pipeline_impl_0$id_ex_illegal
280. = 0
281. pipeline_impl_0$id_ex_opa_select
282. = 00
283. pipeline_impl_0$id_ex_opb_select
284. = 00
285. pipeline_impl_0$id_ex_rd_mem
286. = 0
287. pipeline_impl_0$id_ex_rega
288. = 0000000000000000000000000000000000000000000000000000000000000000
289. pipeline_impl_0$id_ex_regb
290. = 0000000000000000000000000000000000000000000000000000000000000000
291. pipeline_impl_0$id_ex_uncond_branch
292. = 0
293. pipeline_impl_0$id_ex_wr_mem
294. = 0
295. pipeline_impl_0$id_halt_out
296. = 0
297. pipeline_impl_0$id_illegal_out
298. = 0
299. pipeline_impl_0$id_opa_select_out
300. = 00
301. pipeline_impl_0$id_opb_select_out
302. = 00

303. pipeline_impl_0$id_rd_mem_out
304. = 0
305. pipeline_impl_0id_stage_0dest_reg_select
306. = 10
307. pipeline_impl_0id_stage_0fwd_check_EX
308. = 11110
309. pipeline_impl_0id_stage_0fwd_check_MEM
310. = 11110
311. pipeline_impl_0id_stage_0ra_idx
312. = 11111
313. pipeline_impl_0id_stage_0rb_idx
314. = 11111
315. pipeline_impl_0$id_uncond_branch_out
316. = 0
317. pipeline_impl_0$id_wr_mem_out
318. = 0
319. pipeline_impl_0if_stage_0PC_enable
320. = 1
321. pipeline_impl_0if_stage_0PC_plus_4
322. = 0000000000000000000000000000000000000000000000000000000000000100
323. pipeline_impl_0if_stage_0PC_reg
324. = 0000000000000000000000000000000000000000000000000000000000000000
325. pipeline_impl_0if_stage_0next_PC
326. = 0000000000000000000000000000000000000000000000000000000000000100
327. pipeline_impl_0$mem_result_out
328. = 0000000000000000000000000000000000000000000000000000000000000000
329. pipeline_impl_0$mem_wb_dest_reg_idx
330. = 11111
331. pipeline_impl_0$mem_wb_result
332. = 0000000000000000000000000000000000000000000000000000000000000000
333. pipeline_impl_0$proc2Dmem_command
334. = 00
335. pipeline_impl_0$proc2Imem_addr
336. = 0000000000000000000000000000000000000000000000000000000000000000
337. pipeline_impl_0$wb_reg_wr_data_out
338. = 0000000000000000000000000000000000000000000000000000000000000000
339. pipeline_impl_0$wb_reg_wr_idx_out
340. = 11111
341. pipeline_impl_0wb_stage_0result_mux
342. = 0000000000000000000000000000000000000000000000000000000000000000
343. pipeline_spec_0$ex_alu_result_out
344. = 0000000000000000000000000000000000000000000000000000000011001000
345. pipeline_spec_0ex_stage_spec0br_disp
346. = 0000000000000000000000000000000000000000000000000000000011000100
347. pipeline_spec_0ex_stage_spec0id_ex_IR_aux

348. = 010
349. pipeline_spec_0ex_stage_spec0opa_mux_out
350. = 0000000000000000000000000000000000000000000000000000000000000100
351. pipeline_spec_0ex_stage_spec0opb_mux_out
352. = 0000000000000000000000000000000000000000000000000000000011000100
353. pipeline_spec_0$id_alu_func_out
354. = 00000
355. pipeline_spec_0$id_dest_reg_idx_out
356. = 11111
357. pipeline_spec_0$id_opa_select_out
358. = 10
359. pipeline_spec_0$id_opb_select_out
360. = 10
361. pipeline_spec_0$id_rega_out
362. = 1000000000000000000000000000000000000000000000000000000000000000
363. pipeline_spec_0id_stage_spec0dest_reg_select
364. = 10
365. pipeline_spec_0id_stage_spec0ra_idx
366. = 11110
367. pipeline_spec_0id_stage_spec0regf_spec0$rda_reg
368. = 1000000000000000000000000000000000000000000000000000000000000000
369. pipeline_spec_0if_stage_spec0PC_plus_4
370. = 0000000000000000000000000000000000000000000000000000000000000100
371. pipeline_spec_0if_stage_spec0PC_reg
372. = 0000000000000000000000000000000000000000000000000000000000000000
373. pipeline_spec_0if_stage_spec0next_PC
374. = 0000000000000000000000000000000000000000000000000000000011001000
375. pipeline_spec_0$if_stage_spec_0$PC_reg
376. = 0000000000000000000000000000000000000000000000000000000000110000
377. pipeline_spec_0$proc2Dmem_addr
378. = 0000000000000000000000000000000000000000000000000000000011001000
379. pipeline_spec_0$proc2Imem_addr
380. = 0000000000000000000000000000000000000000000000000000000000000000
381. pipeline_spec_0$wb_reg_wr_en_out
382. = 0
383. pipeline_spec_0$wb_reg_wr_idx_out
384. = 11111
385. proc2mem_addr_impl
386. = 0000000000000000000000000000000000000000000000000000000000000xxx
387. proc2mem_command_impl
388. = 01
389. proc2mem_fetch_addr_spec
390. = 0000000000000000000000000000000000000000000000000000000000000xxx
391. spec_mem_idx2
392. = 0000000000000000000000000000000000000000000000000000000000000

393.
394. Cycle 3
395. —————
396. id_ex_IR_impl
397. = 01000111111111110000010000011111
398. if_id_IR_impl
399. = 11101011110000000000000000110001
400. illegal_impl_encountered
401. = 0
402. mem2proc_fetch_data_spec
403. = 10110100000111110000000011001100
404. mem_wb_IR_spec
405. = 10110100000111110000000011001100
406. pipeline_impl_0$ex_mem_alu_result
407. = 0000000000000000000000000000000000000000000000000000000000000000
408. pipeline_impl_0$ex_mem_dest_reg_idx
409. = 11111
410. pipeline_impl_0$ex_mem_halt
411. = 0
412. pipeline_impl_0$ex_mem_illegal
413. = 0
414. pipeline_impl_0$ex_mem_rd_mem
415. = 0
416. pipeline_impl_0$ex_mem_take_branch
417. = 0
418. pipeline_impl_0$ex_mem_wr_mem
419. = 0
420. pipeline_impl_0$id_ex_alu_func
421. = 00000
422. pipeline_impl_0$id_ex_cond_branch
423. = 0
424. pipeline_impl_0$id_ex_dest_reg_idx
425. = 11111
426. pipeline_impl_0$id_ex_fwd_a
427. = 00
428. pipeline_impl_0$id_ex_halt
429. = 0
430. pipeline_impl_0$id_ex_illegal
431. = 0
432. pipeline_impl_0$id_ex_opa_select
433. = 00
434. pipeline_impl_0$id_ex_opb_select
435. = 00
436. pipeline_impl_0$id_ex_rd_mem
437. = 0

438. pipeline_impl_0$id_ex_uncond_branch
439. = 0
440. pipeline_impl_0$id_ex_wr_mem
441. = 0
442. **pipeline_impl_0id_stage_0fwd_check_EX**
443. **= 11111**
444. **pipeline_impl_0id_stage_0fwd_check_MEM**
445. **= 11110**
446. pipeline_impl_0id_stage_0stall_check
447. = 0
448. pipeline_impl_0if_stage_0PC_reg
449. = 0000000000000000000000000000000000000000000000000000000000000100
450. pipeline_impl_0$mem_wb_dest_reg_idx
451. = 11111
452. pipeline_impl_0$mem_wb_halt
453. = 0
454. pipeline_impl_0$mem_wb_illegal
455. = 0
456. pipeline_impl_0$mem_wb_result
457. = 0000000000000000000000000000000000000000000000000000000000000000
458. pipeline_impl_0$mem_wb_take_branch
459. = 0
460. pipeline_impl_0$proc2Dmem_command
461. = 00
462. pipeline_spec_0$ex_alu_result_out
463. = 0000000000000000000000000000000000000000000000000000000011001100
464. pipeline_spec_0ex_stage_spec0mem_disp
465. = 0000000000000000000000000000000000000000000000000000000011001100
466. pipeline_spec_0ex_stage_spec0opa_mux_out
467. = 0000000000000000000000000000000000000000000000000000000011001100
468. pipeline_spec_0ex_stage_spec0opb_mux_out
469. = 0000000000000000000000000000000000000000000000000000000000000000
470. pipeline_spec_0$id_alu_func_out
471. = 00000
472. pipeline_spec_0$id_opa_select_out
473. = 01
474. pipeline_spec_0$id_opb_select_out
475. = 00
476. pipeline_spec_0$id_rega_out
477. = 0000000000000000000000000000000000000000000000000000010101010101
478. pipeline_spec_0$id_regb_out
479. = 0000000000000000000000000000000000000000000000000000000000000000
480. pipeline_spec_0id_stage_spec0ra_idx
481. = 00000
482. pipeline_spec_0id_stage_spec0rb_idx

483. = 11111
484. pipeline_spec_0id_stage_spec0regf_spec0$rda_reg
485. = 0000000000000000000000000000000000000000000000000000010101010101
486. pipeline_spec_0if_stage_spec0PC_reg
487. = 0000000000000000000000000000000000000000000000000000000011001000
488. pipeline_spec_0$proc2Dmem_addr
489. = 0000000000000000000000000000000000000000000000000000000011001100
490. pipeline_spec_0$proc2Dmem_command
491. = 10
492. pipeline_spec_0$proc2Imem_addr
493. = 0000000000000000000000000000000000000000000000000000000011001000
494. proc2mem_addr_spec
495. = 0000000000000000000000000000000000000000000000000000000011001100
496. proc2mem_command_impl
497. = 01
498. proc2mem_command_spec
499. = 10
500. proc2mem_data_spec
501. = 0000000000000000000000000000000000000000000000000000010101010101
502. proc2mem_fetch_addr_spec
503. = 00000000000000000000000000000000000000000000000000000000110001xxx
504. spec_mem_idx
505. = 0000000000000000000000000000000000000000000000000000000011001
506. spec_mem_idx2
507. = 0000000000000000000000000000000000000000000000000000000011001
508.
509. Cycle 4
510. ——————
511. ...
512. Cycle 12
513. ——————
514. f1
515. = 1
516. f2
517. = 1
518. halt_curr
519. = 1
520. halt_prev
521. = 0
522. mem2proc_fetch_data_spec
523. = 00000000000000000000010101010101
524. mem_wb_IR_spec
525. = 000xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
526. mem_wb_NPC_impl
527. = 0000000000000000000000000000000000000000000000000000000000001000

528. pipeline_0if_stage_0PC_reg
529. = 0000000000000000000000000000000000000000000000000000000000110000
530. pipeline_commit_NPC_impl
531. = 0000000000000000000000000000000000000000000000000000000000001000
532. pipeline_error_status_impl
533. = 0010
534. pipeline_impl_0$proc2Dmem_command
535. = 00
536. pipeline_spec_0if_stage_spec0PC_reg
537. = 0000000000000000000000000000000000000000000000000000000011001000
538. pipeline_spec_0$if_stage_spec_0$PC_reg
539. = 0000000000000000000000000000000000000000000000000000000000110000
540. pipeline_spec_0$proc2Dmem_command
541. = 00
542. pipeline_spec_0$proc2Imem_addr
543. = 0000000000000000000000000000000000000000000000000000000011001000
544. proc2mem_command_impl
545. = 01
546. proc2mem_command_spec
547. = 00
548. proc2mem_fetch_addr_spec
549. = 0000000000000000000000000000000000000000000000000000000011001xxx
550. sample_NPC_pipeline
551. = 0000000000000000000000000000000000000000000000000000000011001100
552. spec_mem_idx2
553. = 0000000000000000000000000000000000000000000000000000000011001
554.
555.
556. -I- Updating Lemma DB file: ../work/alphatest.lemmas_db
557. -I- Total iterations: 21
558.
559.
560. Reveal finished.
561.
562. Abst Solving Time: 4.79
563. Total Time: 140.35

Bibliography

[1] M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones, *A framework for microprocessor correctness statements*, Int'l Journal on Software Tools for Technology Transfer (2001), 433–448.

[2] Z. Andraus, M. Liffiton, and K. Sakallah, *Refinement strategies for verification methods based on datapath abstraction*, Proc. of Asia and South Pacific Design Automation Conference, 2006, pp. 19–24.

[3] Z. Andraus, M. Liffiton, and K. A. Sakallah, *CEGAR-based formal hardware verification: a case study*, Technical Report CSE-TR-531-07, University of Michigan, 2007.

[4] Z. Andraus and K. Sakallah, *Automatic abstraction and verification of Verilog models*, Proc. of Design Automation Conference, 2004, pp. 218–223.

[5] Z. A. Andraus, M. H. Liffiton, and K. A. Sakallah, *Reveal: a formal verification tool for Verilog designs*, Proc. of Int'l Conference on Logic for Programming, Artificial Intelligence and Reasoning, 2008, pp. 343–352.

[6] T. Ball and S. Rajamani, *The SLAM project: debugging system software via static analysis*, Annual Symposium on Principles of Programming Languages, 2002, pp. 1–3.

[7] T. Ball and S. K. Rajamani, *Boolean programs: a model and process for software analysis*, Technical Report 2000-14, Microsoft Research, 2000.

[8] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, *Symbolic model checking without BDDs*, Proc. of Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems, 1999, pp. 193–207.

[9] S. Bose and A. Fisher, *Verifying pipelined hardware using symbolic logic simulation*, Proc. of ICCD, 1989, pp. 217–221.

[10] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Staple, G. Swamy, and T. Villa, *VIS: a system for verfication and synthesis*, Proc. of Int'l Conference on Computer-Aided Verification, Lecture Notes in Computer Science 1102, 1996, pp. 428–432.

[11] R. Bryant, S. German, and M. Velev, *Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic*, ACM Transactions on Computational Logic **2** (2001), 93–134.

[12] R. Bryant, S. Lahiri, and S. Seshia, *Modeling and verifying systems using a logic of counter arithmetic with Lambda expressions and uninterpreted functions*, Proc. of Int'l Conference on Computer-Aided Verification, Lecture Notes in Computer Science 2404, 2002, pp. 78–92.

[13] R. E. Bryant, *Graph-based algorithms for boolean function manipulation*, IEEE Trans. on Computers, C-35(8) (1986), 677–691.

[14] R. E. Bryant, S. German, and M. Velev, *Processor verification using efficient decision procedures for a logic of uninterpreted functions*, TABLEAUX, LNAI 1617, 1999, pp. 1–13.

[15] J. Burch and D. Dill, *Automatic verification of pipelined microprocessor control*, Proc. of Int'l Conference on Computer-Aided Verification, Lecture Notes in Computer Science 818, 1994, pp. 68–80.

[16] J. R. Burch, *Techniques for verifying superscalar microprocessors*, Proc. of Design Automation Conference, 1996, pp. 552–557.

[17] A. Camilleri, M. Gordon, , and T. Melham, *Hardware verification using higher order logic*, D. Borrione, editor, From HDL Descriptions to Guaranteed Correct Circuit Designs.

[18] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, *Automated abstraction refinement for model checking large state spaces using SAT-based conflict analysis*, Proc. of Formal Methods on Computer Aided Design, 2002, pp. 33–51.

[19] E. Clarke and O. Grumberg, *Avoiding the state explosion problem in temporal logic model checking*, Proc. of ACM Symposium on Principles of Distributed Computing, 1987, pp. 294–303.

[20] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, *Counterexample-guided abstraction refinement*, Proc. of Int'l Conference on Computer-Aided Verification, 2000, pp. 154–169.

[21] E. Clarke, O. Grumberg, and D. Long, *Model checking and abstraction*, ACM Transactions on Programming Languages and Systems (TOPLAS) **16** (1994), no. 5, 1512–1542.

[22] F. Corella, *Automated high-level verification against clocked algorithmic specification*, Proc. of Conference on Computer Hardware Descriptive Languages and their Applications, 1993, pp. 147–154.

[23] P. Cousot and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, Conference Record of the Sixth Annual ACM SI PLAN-SIGACT Symposium on Principles of Programming Languages, 1977, pp. 238–252.

[24] D. Cyrluk, *Microprocessor verification in PVS: a methodology and simple example*, Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, 1993.

[25] S. Das and D. Dill, *Successive approximation of abstract transition relations*, IEEE Symposium on Logic in Computer Science, 2001, pp. 51–58.

[26] N. Een and N. Sorensson, *An extensible SAT-solver*, Int'l Conference on Theory and Applications of Satisfiability Testing, 2003, pp. 333–336.

[27] M. Gordon, *HOL: A proof generating system for higher-order logic*, G. Birtwhistle and P.A Surahmanyam, editors, VLSI Specication,Verication, and Synthesis (1987), 73–128.

[28] S. Graf and H. Saidi, *Construction of abstract state graphs with PVS*, Proc. of Int'l Conference on Computer-Aided Verification, 1997, pp. 72–83.

[29] A. Gupta, M. Ganai, Z. Yang, and P. Ashar, *Iterative abstraction using SAT-based BMC with proof analysis*, Proc. of Int'l Conference on Computer-Aided Design, 2003, pp. 416–423.

[30] J. Hensessy and D. Patterson, *Computer architecture: A quantitative approach*, 2nd ed., Morgan Kaufman, 1996.

[31] R. Hojati and R. Brayton, *Automatic datapath abstraction of hardware systems*, Proc. of Int'l Conference on Computer-Aided Verification, 1995, pp. 98–113.

[32] R. Hosabettu, M. Srivas, and G. Gopalakrishnan, *Decomposing the proof of correctness of pipelined microprocessors*, Proc. of Int'l Conference on Computer-Aided Verification, Lecture Notes in Computer Science 1427, 1998, pp. 122–134.

[33] H. Jain, D. Kroening, N. Sharygina, and E. Clarke, *Word-level predicate abstraction and refinement for verifying RTL Verilog*, Proc. of Design Automation Conference, 2005, pp. 445–450.

[34] M. Kaufmann, P. Manolios, and J. Moore, *Computer-aided reasoning: An approach*, Kluwer Academic Publishers, 2000.

[35] D. Kroening and E. Clarke, *Checking consistency of C and Verilog using predicate abstraction and induction*, Proc. of Int'l Conference on Computer Aided Design, 2004, pp. 66–72.

[36] R. Kurshan, *Computer-aided verification of coordinating processes: The automata-theoritic approach*, Princeton University Press, 1999.

[37] S. Lahiri, C. Pixley, and K. Albin, *Experience with term level modeling and verification of the M*Core microprocessor core*, Proc. of IEEE Int'l High-Level Design Validation and Test Workshop, 2001, p. 109.

[38] M. Liffiton and K. Sakallah, *Algorithms for computing minimal unsatisfiable subsets of constraints*, Journal of Automated Reasoning **40** (2008), no. 1, 1–33.

[39] P. Manolios, *Correctness of pipelined machines*, Proc. of Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science 1954, 2000, pp. 161–178.

[40] P. Manolios and S. Srinivasan, *Automatic verification of safety and liveness for XScale-like processor models using WEB refinements*, Proc. of Design, Automation, and Test in Europe, 2004, pp. 168–173.

[41] P. Manolios, S. Srinivasan, and D. Vroon, *Automatic memory reductions for RTL model verification*, Proc. of Int'l Conference on Computer-Aided Design, 2006, pp. 786–793.

[42] P. Manolios and S. K. Srinivasan, *Refinement maps for efficient verification of processor models*, Proc. of Design, Automation, and Test in Europe, 2005, pp. 1304–1309.

[43] P. Marques-Silva and K. A. Sakallah, *Grasp - a search algorithm for propositional satisfiability*, IEEE Tran. on Computers, 48(5) (1999), 506–521.

[44] K. McMillan and N. Amla, *Automatic abstraction without counterexamples*, Proc. of Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2003, pp. 2–17.

[45] G. Moore, *Cramming more components onto integrated circuits*, Electronics Magazine, 38(8) (1965), 114–117.

[46] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, *Chaff: engineering an efficient SAT solver*, Proc. of Design Automation Conference, 2001, pp. 530–535.

[47] G. Nelson and D. Oppen, *Simplification by cooperating decision procedures*, ACM Transactions on Programming Languages and Systems **2** (1979), no. 1, 245–257.

[48] Y. Oh, M. Mneimneh, Z. Andraus, K. Sakallah, and I. Markov, *AMUSE: a minimally-unsatisfiable subformula extractor*, Proc. of Design Automation Conference, 2004, pp. 518–523.

[49] S. Owre, N. Shankar, and J. Rushby, *PVS: A prototype verification system*, Proc. of Int'l Conference on Automated Deduction, 1992, pp. 748–752.

[50] J. Sawada and W. Hunt, *Trace table-based approach for pipelined microprocessor verification*, Proc. of Int'l Conference on Computer-Aided Verification, Lecture Notes in Computer Science 1254, 1997, pp. 364–375.

[51] J. Sawada and W. A. Hunt, *Processor verification with precise exceptions and speculative execution*, Proc. of Int'l Conference on Computer-Aided Verification, Lecture Notes in Computer Science 1427, 1998, pp. 135–146.

[52] R. E. Shostak, *Deciding combinations of theories*, Journal of the ACM (JACM) **31** (1984), no. 1, 1–12.

[53] M. Srivas and M. Bickford, *Formal verification of a pipelined microprocessor*, IEEE Trans. on Software Engineering (1990), 52–64.

[54] D. Thomas and P. Moorby, *The verilog hardware description language*, Kluwer Academic Publishers, Nowell, Massachusetts, 1991.

[55] M. Velev, *Microprocessor verification using efficient decision procedures for a logic of equality with uninterpreted functions*, Proc. of Int'l Conference on Automated Reasoning with Analytic Tableaux and Related Methods, 1999, pp. 1–13.

[56] M. Velev., *Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer*, Proc. of Design, Automation, and Test in Europe, 2002, p. 28.

[57] M. Velev and R. Bryant, *Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction*, Proc. of Design Automation Conference, 2000, pp. 112–117.

[58] M. N. Velev., *Automatic abstraction of memories in the formal verification of superscalar microprocessors*, Proc. of Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2001, pp. 252–267.

[59] M. N. Velev, *Automatic formal proof of liveness for pipelined microprocessors*, Journal of Systems Architecture, Special Issue on New Architectures, Methods, and Tools for Circuits and System Synthesis (204), 1–6.

[60] D. Wang, P. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano, *Formal property verification by abstraction refinement with formal, simulation, and hybrid engines*, Proc. of Design Automation Conference, 2001, pp. 35–40.

[61] F. Wang, B. Li, H. Jin, G. Hachtel, and F. Somenzi, *Improving Ariadne's Bundle by following multiple threads in abstraction refinement*, Proc. of Int'l Conference on Computer-Aided Design, 2003, pp. 408–415.

[62] P. Windley and M. Coe, *A correctness model for pipelined microprocessors*, Theorem Provers in Circuit Design **901** (1995), 33–51.

[63] P. J. Windley and J. R. Burch, *Mechanically checking a lemma used in an automatic verification tool*, Proc. of Formal Methods on Computer Aided Design, Lecture Notes in Computer Science 1166, 1996, pp. 362–376.

[64] L. Zhang and S. Malik, *Extracting small unsatisfiable cores from unsatisfiable Boolean formula*, Int'l Conference on Theory and Applications of Satisfiability Testing, 2003.

[65] http://yices.csl.sri.com/.

[66] http://www.kenmcmil.com/smv.html.

[67] Sun Microsystems, ”picoJava-II Microarchitecture Guide”, March, 1999.

[68] http://www.icarus.com/eda/verilog.

[69] http://www.eecs.umich.edu/ zandrawi/reveal/reveal.htm#demo.

[70] http://www.eecs.umich.edu/ zandrawi/reveal/reveal.htm#mips.

[71] http://www.eecs.umich.edu/ zandrawi/reveal/reveal.htm#sorter.

[72] http://www.eecs.umich.edu/ zandrawi/reveal/reveal.htm#icram.

[73] http://www.eecs.umich.edu/ zandrawi/reveal/reveal.htm#omu.

[74] http://www.eecs.umich.edu/ zandrawi/reveal/reveal.htm#dlx.

[75] http://www.eecs.umich.edu/ zandrawi/reveal/reveal.htm#risc.

[76] http://www.eecs.umich.edu/ zandrawi/reveal/reveal.htm#x86.

[77] http://vlsi.cs.iitm.ernet.in/x86_proj/x86Homepage.html.

[78] http://www.opencores.org.

[79] http://www.cs.cmu.edu/ uclid.

[80] http://www-static.cc.gatech.edu/fac/Pete.Manolios/bat/.

[81] http://www.cs.cmu.edu/ modelcheck/vcegar/.

[82] http://vlsi.colorado.edu/ vis/.

[83] http://www.princeton.edu/ chaff/.