The Dissertation Committee for Shilpi Goel
certifies that this is the approved version of the following dissertation:

Formal Verification of Application and System Programs Based on a Validated x86 ISA Model

Committee:

Warren A. Hunt, Jr., Supervisor

Lorenzo Alvisi

Matt Kaufmann

Calvin Lin

J Strother Moore

Robert Watson

Formal Verification of Application and System Programs Based on a Validated x86 ISA Model

by

Shilpi Goel, B. Tech.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

The University of Texas at Austin

December 2016

To Ma, Du, and Di —

the medical doctors of the family

Acknowledgments

I thank my supervisor, Warren Hunt, Jr., for his guidance and encouragement. He has always given me the freedom and the opportunities to pursue my research interests. I have learned a lot from conversations with him about research, industry, and life in general. Words fall short to express my gratitude to him. I thank Matt Kaufmann for being a wonderful mentor and a patient teacher. His unceasing interest in the progress of this research has been a constant source of motivation. I thank J Moore for offering suggestions for dealing with large interpreter-based proofs, and for being generous with his time. I thank Robert Watson for offering detailed feedback on this research as well as its presentation. I thank Lorenzo Alvisi and Calvin Lin for their time, expertise, and feedback. I am honored to have such illustrious members of the Computer Science community on my dissertation committee.

I thank the people I worked with at Centaur Technology, including Anna Slobodova, Jared Davis, and Sol Swords. Working with them has been a delightful experience. I am also grateful to Centaur for providing me the opportunity to visit the University of Cambridge, where I interacted with many other brilliant people.

I thank Marijn Heule for his feedback on this dissertation and for countless fun conversations about everything under the sun. I will always be indebted to him for his support and advice. I am in awe of the creative genius of Arjen van Lith, who has my undying gratitude for designing the logo of `x86isa`, the ACL2 library that is

a product of this dissertation.

Nathan Wetzler has been a marvelous friend and colleague, who always made time to discuss technical issues or simply to catch a movie when I needed a break. David Rager has offered many much-needed words of encouragement over the years. I would like to thank other former and current members of the ACL2 research group at UT, including Robert Krug, Bill Young, Cuong Kim Chau, Soumava Ghosh, Keshav Kini, Ben Selfridge, and Mihir Mehta. I consider myself fortunate that I got the chance to exchange ideas with the people in this group.

I thank Robert Krug, Cuong Kim Chau, and Dmitry Nadezhin for their contributions in the `x86isa` library. I am grateful to Eric Smith for his feedback on this dissertation and for giving me the opportunity of interacting with the wonderful people at Kestrel Institute.

I thank Defense Advanced Research Projects Agency (DARPA) and National Science Foundation (NSF) for their support. The work presented in this dissertation was supported by DARPA under contract number N66001-10-2-4087, and by NSF under contract number CNS-1525472. I am also grateful to Lindy Aleshire for helping me with the logistics of being an international graduate student at UT.

My friends have helped me keep things in perspective these past years — thanks for all the conversations, hikes, road trips, and culinary adventures.

Finally, I thank my family for always being there for me, despite the geographical distance between us. My parents make it incredibly easy to love, like, and admire them, and they have always inspired me at every walk of my life.

Formal Verification of Application and System Programs Based on a Validated x86 ISA Model

Publication No. _____

Shilpi Goel, Ph.D.
The University of Texas at Austin, 2016

Supervisor: Warren A. Hunt, Jr.

Two main kinds of tools available for formal software verification are point tools and general-purpose tools. Point tools are targeted towards bug-hunting or proving a fixed set of properties, such as establishing the absence of buffer overflows. These tools have become a practical choice in the development and analysis of serious software systems, largely because they are easy to use. However, point tools are limited in their scope because they are pre-programmed to reason about a fixed set of behaviors. In contrast, general-purpose tools, like theorem provers, have a wider scope. Unfortunately, they also have a higher user overhead. These tools often use incomplete and/or unrealistic software models, in part to reduce this overhead. Formal verification based on such a model can be restrictive because it does not account for program behaviors that rely on missing features in the model. The results of such formal verification undertakings may be unreliable — consequently, they can offer a false sense of security.

This dissertation demonstrates that formal verification of complex program properties can be made practical, without any loss of accuracy or expressiveness, by employing a machine-code analysis framework implemented using a mechanical theorem prover. To this end, we constructed a formal and executable model of the x86 Instruction-Set Architecture using the ACL2 theorem-proving system. This model includes a specification of 400+ x86 opcodes and architectural features like segmentation and paging. The model's high execution speed allows it to be validated routinely by performing co-simulations against a physical x86 processor — thus, formal analysis based on this model is reliable. We also developed a general framework for x86 machine-code analysis that can lower the overhead associated with the verification of a broad range of program properties, including correctness with respect to behavior, security, and resource requirements. We illustrate the capabilities of our framework by describing the verification of two application programs, *population count* and *word count*, and one system program, *zero copy*.

Table of Contents

List of Figures

Part I

Preliminaries

Chapter 1

Introduction

Almost every aspect of modern society — including critical areas like defense, finance, transportation, and health care — relies heavily on software systems. The use of faulty software can have grim consequences, ranging from inconveniences like service interruptions to catastrophes that jeopardize human life. Verifying that programs behave as expected during run-time is of paramount importance. However, thorough verification of software is considered impractical, largely due to the user effort required in such undertakings.

Our thesis is that mechanical verification of complex program properties can be made practical, without any loss of accuracy or expressiveness, by employing a machine-code analysis framework implemented using a theorem-proving system. In defense of this thesis, we developed a general-purpose analysis framework for x86 machine-code programs in the ACL2 theorem-proving system [19, 134], and used it to reason mechanically about properties of real programs, including correctness with respect to behavior, security, and resource requirements. Our framework has been designed with practicality in mind, and as such, it offers various features to enable different depths of analysis, depending on the kinds of programs being considered for verification and the goals of the verification effort.

Though program verification has a long history, much of the research in this area has resulted in tools that facilitate bug-hunting or proving a fixed set of program properties, such as establishing the absence of buffer overflows. Program analysis is often carried out using incomplete and/or unrealistic models of the environment. The danger of working with such models is that even if a program has been formally verified with respect to a specification, there is no guarantee that it will behave as dictated by that specification in real life. For example, a program proven to be correct under the simplifying assumption that the virtual memory abstraction remains intact during its execution might suffer from mysterious crashes in real life if the virtual memory manager is buggy — for instance, it may allow other processes to overwrite the program or its data. Thus, analysis performed using simplifying or implicit assumptions can be misleading, because it can purport to offer a higher degree of assurance than it actually does.

Considering the wide variety of functionality expected from computer programs, we need practical, general-purpose tools to analyze program behavior accurately — that is, to perform *heavyweight* formal software verification. These tools must be capable of accepting user guidance effectively, given the undecidable nature of verifying program correctness. Theorem provers fit the bill of such interactive formal tools. However, theorem provers are infamous for requiring time-consuming manual labor to complete proofs, and this social reputation deters many serious practitioners. Hybrid approaches that combine interactive and automatic tools are gaining traction, but completely automatic approaches, restricted either in the kinds of properties they can verify or the state space they can cover, are still preferred for

commercial program verification because they impose a smaller burden on the users.

The goal of this research is to make formal analysis of machine code a practical choice in the development and verification of serious software systems. We believe that our general-purpose analysis framework for x86 machine-code programs constitutes a significant step in this direction. In the rest of this dissertation, we use the term "programmer" to refer to a computer software developer, and the term "user" to refer to a formal software verification practitioner. Of course, these two terms can refer to the same person once heavyweight formal analysis of software becomes practical.

We present an overview of our dissertation project in Section 1.1, and summarize the contributions of this research in Section 1.2. In Section 1.3, we describe the organization of this dissertation and also present a recommended reading order based on the interest of the reader.

1.1 Overview

Though using high-level semantics for program verification may seem appealing, we advocate performing verification at the machine-code level. There are several downsides to high-level program analysis, which we discuss in detail later in Section 2.2 but summarize here. Analysis frameworks for some high-level languages simply do not exist. The semantics of high-level languages such as C are ambiguous, even though they have been standardized by ISO, whereas the semantics of machine code are comparatively better defined. High-level program analysis cannot be used directly in situations where only machine-code programs are available

(e.g., malware, executables downloaded from the Internet, low-level firmware and OS code, etc.). Moreover, the applicability of analysis of high-level programs depends on the correctness of compiler transformations. Unfortunately, compilers are large software systems that evolve frequently and whose correctness is non-trivial to ascertain. Also, verified and verifying compilers for most high-level languages of interest are not mainstream yet.

Therefore, it is prudent to develop capabilities to analyze machine code for commercially available processors. A big benefit of machine-code analysis tools is their universal applicability — they can be used to verify all programs that can compile down to the supported hardware platform[1]. This research focuses on the x86 ISA, which is the dominant processor architecture for non-embedded devices. Also, because the x86 ISA is one of the more complex modern architectures, insights gained over the course of this dissertation project will be applicable when working with other architectures.

This dissertation discusses three main topics:

1. A formal and executable model of the x86 instruction set architecture (ISA) written in ACL2's logic — this model contains the specification of both user- and system-mode instructions and features, thereby providing formal semantics to x86 machine-code programs;

2. General-purpose program analysis libraries that reduce the time and manual

[1]This is true as long as the formal model for machine-code analysis specifies all the features that interact with a given program in any way.

effort required to prove properties of x86 machine-code programs — together, the x86 ISA model and these libraries form our x86 machine-code analysis framework;

3. Case studies of application and system programs that demonstrate the capabilities of our framework and present verification strategies that can be adopted to verify a variety of machine-code programs.

1.1.1 x86 ISA Model

The focus of this research is to specify and verify 64-bit x86 programs. Consequently, our x86 ISA model specifies the 64-bit sub-mode of Intel's IA-32e mode [36]. Major contemporary operating systems, such as Windows, Linux, FreeBSD, and Mac OS, operate in this mode of Intel processors. Our x86 model specifies a single-processor x86 architecture and models 400+ machine opcodes, including supervisor mode and floating-point instructions. It includes the specification of the x86 memory management via paging and segmentation. Our x86 model is sufficiently complete to support the specification and verification of most machine code programs emitted by mainstream compilers like GCC [32] and LLVM [47].

Conceptually, our approach to modeling the x86 ISA is straightforward. Our model includes an x86 state that characterizes state components of the ISA, like registers, flags, and memory. A transition function transforms the current x86 state to the next one by taking one step, i.e., executing a single x86 instruction. Simulation of an x86 machine-code program is accomplished by taking repeated steps. It is imperative that our specification of the x86 ISA be faithful to the real machine —

the applicability of the results of formal analysis depends on the accuracy of the x86 model. However, the sheer size and complexity of the x86 ISA makes the development of its model difficult. The x86 ISA is described by Intel manuals [40] that are large documents (around 3500 pages), consisting mostly of English prose. Developing an x86 model is an error-prone process, subject to interpretation of the English text in these manuals, though ambiguities can sometimes be resolved by cross-referencing the AMD manuals [21] and running tests on physical x86 machines.

Given the error-prone nature of model development, the directly executable aspect of our model plays a crucial role in this project — it facilitates model validation via co-simulations against real x86 machines and emulators like QEMU [90]. Another benefit is that our model can be used as an instruction-set simulator, which allows inspection of the behavior of programs using the classic way of running concrete tests in a safe environment. The simulation speed of our model is either around 320,000 instructions/second or 3.3 million instructions/second, depending on its mode of operation[2]. To our knowledge, this is the fastest formal simulator for an ISA as complex as the x86.

1.1.2 Program Analysis Libraries

Building the x86 model was a preliminary step, albeit a necessary and challenging one, to enable software analysis. We also developed a general-purpose framework to support *symbolic simulation* of x86 machine-code programs on the x86 model. That is, a final (or next) x86 state can be described in terms of updates made to

[2]This was measured on a machine with Intel Xeon E31280 CPU @ 3.50GHz with 32GB RAM.

the initial (or current) x86 state. These updates can write either symbolic or concrete values to the components in the x86 state, thereby allowing consideration of many, if not all, possible executions at once. We can choose to project out only those parts of an x86 state that are relevant to the property under scrutiny. An illustrative example is as follows. If we needed to verify that a successful run of a given machine-code program stores the Fibonacci number corresponding to its input `n` in the x86 register `rax`, we would project out `rax` from the final x86 state. Then, we would prove that, under a certain set of conditions, the symbolic value at `rax` is the same as that computed by a simple Fibonacci specification function on the same input `n`, i.e., the behaviors of the specification function and the x86 program match. Note that the specification can be a recursive Fibonacci function (recurrence relations are a common mathematical way of defining the Fibonacci function) and the program can implement Fibonacci using an optimized algorithm, e.g., one that uses memoization.

Our lemma libraries can facilitate symbolic simulation of programs with little to no user intervention. We also describe ways to determine the cause of breakdowns, if any, in the automation of symbolic simulation, and possible solutions to enable efficient reasoning.

1.1.3 Case Studies

This dissertation presents the analyses of some programs that were done by employing our symbolic simulation framework. We discuss the verification of two application programs (i.e., programs running in the user-space with few privileges)

in detail: *pop-count* and *word-count*. Pop-count or population count is a straight-line program that uses complex bit-vector operations to compute the number of 1s in the binary representation of an integer in a non-obvious manner. The word-count program makes system calls, which are non-deterministic from the point of view of an application program, to read input from the user. This program computes the character, word, and line counts in this input. We also present our analysis of a system program (i.e., a program running in the kernel-space or supervisor mode) called the *zero-copy*. This program copies data from one linear memory location to another disjoint linear memory location by implementing a *copy-on-write* technique — instead of actually copying the data, it modifies the page table of the destination linear addresses so that it points to the same physical addresses as the source linear addresses.

The programs for these case studies were strategically chosen to demonstrate that our analysis framework can be used to reason mechanically about different kinds of programs using different kinds of techniques. The pop-count program was verified completely automatically using a pre-existing bit-blasting library. Its analysis illustrates that our framework can use automated tools to reduce the proof burden on the user. Analysis of the word-count program involved modeling and reasoning about system calls. This demonstrates that our framework is capable of reasoning about non-determinism. Analysis of the zero-copy program involved reasoning about accesses and updates made to the paging data structures, which are responsible for maintaining the linear memory abstraction. This analysis illustrates that our framework can be used to reason about supervisor-mode programs that modify critical

system data structures.

1.2 Contributions

This research focuses on developing capabilities to state and verify complex program properties and to make program verification via machine-code analysis practical. The contributions of this research are as follows.

- We developed an accurate formal model of the x86 ISA that serves as a compile-to specification for program verification and an unambiguous reference for the x86 ISA. No simplifications of the semantics of the x86 ISA were made.

- We developed a general-purpose framework for the analysis, both testing-based and formal, of x86 application and system programs. This research offers insight into performing program verification taking ISA features into account. This is especially useful when verifying system programs, like kernel code, where low-level ISA features are directly accessible by software.

- This dissertation documents the engineering aspects of building a large-scale formal analysis framework. Though developing and improving verification strategies is critical to ensure software reliability, the nature of this research — making machine-code analysis practical — suggests that it is equally important to examine the engineering decisions driving our framework's development.

Our x86 machine-code analysis framework is available freely under a permissive 3-Clause BSD license [51] and its documentation is also accessible online [50].

The following publications describe some parts of the research presented in this dissertation:

- Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann. Abstract Stobjs and Their Application to ISA Modeling. In *Proceedings of the ACL2 Workshop 2013, EPTCS 114, pp. 54-69*, 2013

- Shilpi Goel and Warren A. Hunt, Jr. Automated Code Proofs on a Formal Model of the x86. In *Verified Software: Theories, Tools, Experiments (VSTTE'13)*, volume 8164 of *Lecture Notes in Computer Science*, pages 222–241. Springer Berlin Heidelberg, 2014

- Shilpi Goel, Warren A. Hunt, Jr., Matt Kaufmann, and Soumava Ghosh. Simulation and Formal Verification of x86 Machine-Code Programs That Make System Calls. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design (FMCAD'14)*, pages 18:91–98, 2014

- Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann. Engineering a Formal, Executable x86 ISA Simulator for Software Verification. In *Provably Correct Systems (ProCoS)*, 2015

In addition to the direct contributions listed above, this research motivated various improvements and feature additions to the ACL2 system, thereby benefiting the ACL2 and theorem proving community at large. A notable example is abstract stobjs, discussed later in this dissertation.

1.3 Organization

The rest of this dissertation is organized as follows.

Part I: Preliminaries gives an introduction to this research, along with some background.

Chapter 2 discusses previous work related to this research.

Chapter 3 examines the design goals that our x86 machine-code analysis framework aspires to achieve, and it presents an overview of our approach to building the x86 ISA model and using it to specify the behavior of programs.

Chapter 4 introduces the ACL2 theorem-proving system, including some features used extensively in our project. It also discusses why ACL2 is well-suited for developing our x86 machine-code analysis framework.

Part II: x86 ISA Model describes our formal, executable specification of the x86 instruction-set architecture in detail.

Chapter 5 presents the components of the x86 ISA state supported by our model, and describes how the state has been defined to support both formal reasoning and execution efficiency.

Chapter 6 introduces the two main modes of operation of our x86 ISA model from the point of view of their memory models.

Chapter 7 describes some challenges we faced while capturing the behavior of x86 instructions, specifically non-deterministic computations, and our solutions to overcome these challenges.

Chapter 8 describes our x86 ISA interpreter, and presents our co-simulation framework that is used for model validation.

Part III: Program Analysis Libraries concerns our general-purpose libraries that enable mechanical reasoning about programs.

Chapter 9 presents some typical lemmas contained in our libraries, and describes our general strategy to reason about machine-code programs using these lemmas.

Chapter 10 describes the special considerations and techniques needed when analyzing supervisor-mode programs.

Part IV: Case Studies demonstrates the capabilities of our framework by means of three case studies that involve analyzing real x86 programs.

Chapter 11 presents the formal analysis of two application programs — pop-count and word-count.

Chapter 12 presents the formal analysis of a supervisor-mode program — zero-copy.

Part V: Epilogue summarizes this research and presents some concluding remarks.

Chapter 13 discusses our conclusions and potential for future research.

1.3.1 How to Read this Dissertation

Though the recommended reading order for the chapters in this dissertation is the same as the order in which they appear, we recognize that a reader may be more interested some topics than others. We make some suggestions to aid the reader in this regard.

Chapter 3 presents a broad overview of this dissertation project without going into specifics. The reader is strongly urged to read this chapter so as to concretize their idea of the scope of this dissertation.

Part II of this dissertation focuses mainly on the engineering process of specifying a system as large and complex as the x86 ISA such that it enables both formal reasoning and efficient execution. A reader interested in developing practical formal models and balancing the disparate demands placed on such models is encouraged to read this part.

Part III presents techniques to reason about programs using our x86 ISA model and lemma libraries. A reader interested in developing tools to prove arbitrary properties of programs can choose to focus on this part of the dissertation.

Part IV is recommended for a reader who wishes to assess the capabilities of our analysis framework by observing it "in action", i.e., by studying how it can be applied to prove properties of programs.

Chapter 2

Related Work

The motivation for this research is to enable formal analysis of programs without compromising on precision and practicality. We accomplish this by building a formal, executable model of the x86 ISA using the ACL2 theorem-proving system and developing general-purpose code proof libraries to facilitate x86 machine-code analysis. Our research extends the state-of-the-art by building upon prior work on formal analysis of large computing systems. In this chapter, we discuss some notable works that are relevant to our research and/or share our goals.

2.1 Machine Models

Building models of large computing systems is a standard practice — these models are used to perform both pre- and post-deployment analysis. For example, formal models of processor ISAs have long been used as a target specification for microprocessor design verification [55, 84, 113, 153, 181, 184]. Rockwell Collins built a symbolic simulator [81] for their JEM1 microprocessor in the PVS theorem proving system [161] to detect microcode errors. Another Rockwell Collins project used ACL2 to formally verify that their AAMP7G microcode implements the EAL 7 standard security specification [186]. Though such hardware-related efforts share some of the

same concerns as ours (e.g., accurate model building), in this section the focus is on machine models that were developed in a mathematical logic, and were used or are intended to be used for formal verification of software.

The following projects have directly influenced our x86 ISA modeling approach:

- The CLI stack project [188] of the late 1980s consisted of a stack of mechanically verified systems in NQTHM [68], a predecessor of the ACL2 theorem prover. This stack included a gate-level microprocessor design called FM9001 [180], an assembler for an assembly language called Piton [104] that targeted FM9001, and a higher-level language called micro-Gypsy [193] that targeted Piton. These systems were specified by formal interpreters written using operational semantics. Each of these interpreters was validated and used to prove the correctness of the system above it. The CLI stack, though composed of systems that are much simpler than modern ones, set a milestone in the history of designing and verifying computing systems, from software all the way down to hardware. This project inspired many other undertakings, with the European ESPRIT Provably Correct Systems (ProCoS) projects [109], aimed to study and develop techniques for building correct systems, being one of the more prominent ones.

- Boyer and Yu formalized most of the user-mode instruction set of the Motorola MC68020 microprocessor [158] in NQTHM. This formal model was used

to verify Motorola machine code produced by compiling the Berkeley string library.

- The Java Virtual Machine (JVM) was formalized [98] in ACL2 in order to reason about Java programs at the level of JVM bytecode. This JVM model was optimized for execution efficiency using many techniques published in engineering studies [83, 135] about building formal models in ACL2 that can also serve as efficient simulators.

As in each of the projects above, our x86 ISA model is specified using an interpreter-style operational semantics [102] (more details in Chapter 3). Our model includes the x86 user-mode instruction set as well as supervisor-mode instructions and features. Like the JVM project, our work advocates software analysis via formal verification of low-level code and aims to build formal models that can serve as efficient simulators. As far as we know, our x86 ISA model is the largest and most efficient machine model specified using a theorem prover.

Building accurate ISA models is a challenging endeavor, and domain-specific languages [60, 91, 143, 171, 179] have been developed in order to facilitate clear and precise specification of ISAs, by experts and non-experts alike, to reduce the possibility of introducing errors.

The CHERI (Capability Hardware Enhanced RISC Instructions) ISA [155] is a new architecture that aims to provide software compartmentalization by supporting a hybrid capability model [122] at the level of the processor itself. The CHERI project has been designed with formal verification in mind, and as such, formal models of its

ISA have been developed using L3 [91], a domain-specific language for instruction-set descriptions, and PVS. These formal models are intended to be used for both software and hardware verification.

Formal specifications of processors have been done in other theorem proving systems like Isabelle/HOL [93, 177] and Coq [30, 46]. Commercial multiprocessors like ARM [59, 61], PowerPC [106, 164], and x86 [148, 150, 174] have been modeled in the HOL theorem prover to capture their memory-ordering semantics, given a relaxed-memory concurrency model. The focus of our dissertation is different in that it aims to develop an analysis framework for the mechanical verification of x86 machine-code programs. Our x86 ISA specification models a uniprocessor x86 machine and thus, it provides a sequentially consistent memory model. It can be extended to reason about multi-threaded or multiprocess programs as a part of future work (see Chapter 13). Another formalization of the ARM instruction set and addressing modes [191] has been done using the Coq proof assistant. This ARM model provided instruction specifications that were used to validate the ARM instruction implementations provided by SimSoC (a System-on-Chip simulator) [99]. There is limited support for symbolic simulation of ARM machine code using this formal model. Morrisett has been focused on building scalable formal models for reasoning [94]. Shao's recent efforts to develop and certify clean-slate OS kernels [166] have involved modeling processor architectures in Coq.

These formal specifications mentioned above are not directly executable. Executable definitions have to be extracted from the formal logic in order to perform model validation via co-simulations. This extraction process is non-trivial and thus,

the user has to either trust or verify that the extraction produces executable definitions that are functionally equal to the formal specifications. A benefit of using ACL2 as our choice of theorem prover is that its logic is directly executable — there is no possibility of any cognitive dissonance because the same functions are used for both model validation and reasoning.

Prior to developing our x86 ISA model, we used the Y86 [152] as a prototype. The Y86 is a simple 32-bit x86-like processor developed by Bryant and O'Hallaron for pedagogical purposes. We implemented new architectural and design features in the pre-existing ACL2-based Y86 model, and ported them over to our more complicated x86 ISA model only after their efficacy was determined. Like our x86 model, this Y86 formalization is available freely as a part of the ACL2 Community Books [9].

2.2 Program Analysis

Static and dynamic analysis tools [25, 31, 35, 54, 108, 147] for software bug-hunting are available commercially. These tools have been successful in carrying out their particular functions, like detecting memory safety violations [64, 120, 194]. Also, many of these tools do not take specifications as input — instead, they have built-in "implicit" specifications. Such tools often operate fully automatically, thereby precluding any user guidance. This limits the kinds of properties that such tools can verify. The focus of this dissertation is to build capabilities to mechanically verify arbitrarily complex properties of programs.

Most program analysis frameworks are targeted at high-level languages [75, 76, 89, 118, 160]. These frameworks cannot be used directly in situations where only a

machine-code program is available (e.g., malware such as viruses, worms, and trojans, executables downloaded from the Internet, low-level firmware code). Moreover, the assurances obtained by verifying high-level programs are only valid if the compiler's transformations are correct. Compiler bugs can be resolved by developing verified compilers [62, 77, 136, 187] or verifying compilers [48, 73]. Work has also been done to verify compiler transformations for concurrent programs [71]. Such compilers are not mainstream tools yet because they offer considerably less performance than their counterparts like GCC or LLVM. That being said, CompCert [29, 190] is improving the state-of-the-art: it is a verified compiler for a subset of C that targets ARM, PowerPC, and 32-bit x86 processors — however, it is not available for 64-bit x86 machines yet. A downside of using verifying/verified compilers is that one would need to develop them for every source language of interest, while machine-code verification applies whenever the high-level language has a compiler targeting that processor. Also, even when verified/verifying compilers are used, high-level program analysis cannot guarantee that a program does not access or modify unauthorized memory locations. This is because programs rely on low-level operating system code for services like memory management [43, 44, 115, 128], and often, such low-level code is written in assembly language. Another drawback of high-level language analysis is that the semantics of these languages are often unclear and under-specified. For example, C, widely used for both system and application programming, is notorious for exhibiting ambiguous behavior, though there have been studies [114] to demystify the properties that can be assumed of C code. On the other hand, the semantics of machine code are comparatively more well-defined because processor vendors use

the ISA as a target for their micro-architecture implementations.

Serious interest in low-level code verification arose only in the last three decades, even though program verification has a long history, with Turing's 1949 paper [178] being one of the earliest works on program correctness. Clutterbuck and Carré were among the first to advocate low-level code verification [78]. They analyzed Intel 8080 machine code by using a proof checker to discharge proof obligations generated by a Verification Condition Generator (VCG). Bevier developed a simple multitasking operating system kernel for a von Neumann machine and performed machine-code analysis to verify some of the services offered by this kernel [65] using the NQTHM theorem prover. Boyer and Yu [158] verified user-level machine-code programs in NQTHM using a formal model of a Motorola MC68020 microprocessor. Even though that project involved time- and effort-intensive proofs, it served as a landmark in the history of machine-code verification performed using theorem-proving techniques.

Attempts to reduce the overhead associated with low-level verification have been made by employing compositional verification techniques and specialized Hoare-style logics. Matthews et al. mechanized assertional reasoning by implementing a VCG in ACL2, and made heavy use of compositional reasoning to verify Java bytecode programs [111]. Feng et al. used the Coq proof assistant [30] to verify machine code on a simplified formal model of the x86 processor using domain-specific and separation logics [192]. Myreen's *decompilation into logic* technique [125, 127] reduces the problem of reasoning about machine code to reasoning about simpler logic functions. Decompilation takes machine code as input, and produces logic func-

tions capturing the functional behavior of machine code and a theorem that relates the machine code to these functions. This technique has been used in the Jitawa project [126], to produce a verified Lisp runtime for Milawa [107], a self-verifying theorem prover. Decompilation can be used for verifying machine code on different architectures [144]. So far, this technique has been used for the verification of user-level programs, and does not (yet) have a reasoning methodology for non-deterministic behavior or address translations via paging. Moore has developed a tool called Codewalker [3] in ACL2, which also implements decompilation — it allows formal exploration of code in any programming language that has been specified by an ACL2 model. In addition to developing strategies to reduce user overhead involved in formal verification of machine code, this research is concerned with increasing confidence in the *results* of formal analysis. We discuss how we accomplish this in Chapter 8, where we describe the process for validating our x86 ISA model.

Some other efforts to analyze machine-code programs include work done by Morrisett et al. in software fault isolation [95]. This entailed developing a Coq-based x86 ISA specification that can be used for machine-code verification. This specification is not directly executable—an executable OCaml simulator (with an execution rate of around 50 instructions/second) has to be extracted from the Coq code. Reps et al. [123] developed a sophisticated system, TSL, that can create re-targetable tools for different types of data flow analyses on machine code. Proof-Carrying Code (PCC) [145, 146] and Typed Assembly Language [142] are used to obtain some fixed safety properties (like type safety) of low-level code.

We now discuss efforts targeted towards supervisor-mode code verification.

Perhaps the most notable work here is that for seL4 [45, 119], the "world's first operating-system kernel with an end-to-end proof of implementation correctness and security enforcement". The initial proof [92] of functional correctness of seL4 was about the C source code semantics in Isabelle/HOL. Recently, the ARM machine-code program obtained by compiling seL4 has been proved to be a refinement of the semantics of its C source code [176]. Another related work in this area is that of the Verisoft project [57], which presents the verification of a microkernel targeted for Verified Architecture Microprocessor (VAMP) machines [66]. Shao et al. focus on clean-slate development of OS kernels [166, 195] by employing PCC and modular verification. Though related, our focus is different from that of all these projects. This dissertation project aims to develop a formal, executable ISA model of a commercially-available processor that provides the basis of a machine-code analysis framework. The emphasis of our research is on developing tools and techniques for the verification of existing software. A significant part of our work is about engineering choices that formal tool developers make in order to enable such large-scale verification efforts.

Later in this dissertation (see Chapters 6 and 10), we describe our specification of the IA-32e memory management (paging and segmentation), and discuss how we provide capabilities to reason about programs that make accesses and updates to the paging data structures. A related effort is by Alkassar and others, who formulated properties of "x64-like" paging data structures and TLB, and proved shadow page table algorithms correct [56] using VCC [89]. Another relevant work is by Kolanski, who used the Isabelle/HOL theorem prover to reason about an ARMv6

supervisor-mode program that modifies the virtual memory abstraction [151]. Kolanski developed his own extension of Separation Logic [110], called the *Typed Mapped Separation Logic*, in order to provide a unified framework to reason about both physical and virtual memory. The specifications used in this work do not capture page tables in their entirety; for example, the permissions field of these data structures have been left out. However, the author states that adding support for such missing fields is not hard. Both these works were done at the level of C language. Dahlin et al. used ACL2 to prove that the nested page tables set up by a simple hypervisor, MinVisor, had the desired properties [140]; this work was done using an extended model of the Y86 [152].

The more general problem of reasoning about data structure reads and writes has received considerable attention in the ACL2 community itself. Greve and Wilding had issued a challenge to the ACL2 users to devise efficient ways to reason about complex and pointer-rich dynamic data structures in a linear address space [82]. Specifically, they called for efficient solutions for proving non-interference properties of data structures. An example of a research question they posed was: does a non-interference proof scale quadratically with the number of entries in the data structure, and if so, can we do better? Sophisticated solutions were posted to answer such questions. Moore used *memory taggings* [103] to mark how each memory address of interest will be interpreted by accessor and updater functions. These tags were then used to formulate and prove non-interference theorems. Liu took the approach of proving that accesses and updates do not affect the format and sizes of the data structures under consideration, and lifted reasoning about on-the-fly updates to

a sequence of updates in order to separate traversals from explicit modifications [97]. Greve used *address enumeration* [79], supported by libraries that aid in specification of memory regions [88], to collect a set of all the addresses that specify a data structure so that disjointness properties can be stated and proved about these memory locations of interest. More recently, Wetzler developed a data structure called `farray` that provides efficient execution and convenient theories for reasoning about data accesses and updates [185]. In our work, we build upon some of these solutions to reason about data structures embedded in both linear and physical address spaces.

Chapter 3

Approach

Inspection of the behavior of machine-code programs is often done by employing instruction-set simulators like QEMU [90] and Bochs [116] to run tests. These tools can achieve high simulation speeds, which allows performing a large number of tests in a short amount of time. However, it is infeasible to cover all the possible cases using testing. A solution is to build a mathematical model of machine code that describes its behavior, and then prove theorems about this model in order to establish properties of the machine code. We subscribe to this technique, which falls under the realm of *formal methods*.

One can argue that these simulators do describe the behavior of machine code — after all, they are capable of simulating machine-code programs. Can they then not be used as models for verifying properties of machine code? The answer to this question is yes, but only for those simulators written in a formal specification language. In order to facilitate the use of formal methods, a model has to be *mathematical*. Such a model needs to be developed in a formal specification language, which provides three crucial components — syntax, semantics, and rules to infer new information from known facts (or definitions)[1]. Unfortunately, practical instruction-

[1]See this paper [121] by Lamsweerde that provides an excellent overview of formal specifications.

set simulators are written in languages like C and C++ for execution efficiency, and these languages do not provide an infrastructure for formal analysis.

One can also argue that proving theorems about a formal model does not necessarily mean that they are applicable to the real system — after all, the model may not be faithful to the system under consideration. How do we ensure that our formal model is accurate? This is a legitimate concern, and one that exists in many disciplines (consider architectural models used to evaluate the strength of buildings, protein structural models used to predict the interaction among different kinds of proteins, etc.). One solution is to ask different people to perform reviews of the formal model. This can only take us so far, especially if the system, and consequently the model, is large and complex. Another solution is to perform *co-simulations* — that is, run tests on the model and compare the results to those of the same tests performed on the real system. Co-simulations can also be performed against systems like QEMU and Bochs, though this means that we would have to assume that these systems are faithful to the real processor. Clearly, the more co-simulations we perform (both in diversity and in number), the higher our confidence in the model's accuracy. Unfortunately, most formal specification languages are not known for their execution efficiency. These languages are tailored to describe the "what" of computation, not the "how". Consequently, some of them are not even directly executable, and those that are indeed executable, do not offer efficient execution.

Thus, ideally, we need a formal, executable instruction-set simulator for machine-code programs that can not only describe their behavior mathematically, but

can also allow co-simulations. This chapter presents an overview of our x86 machine-code analysis framework, centered around our x86 ISA model, that satisfies these requirements and enables reliable program analysis. Our framework is written using the ACL2 theorem proving system [19, 134], which allows the definition of logical functions that can be executed efficiently. In Section 3.1, we describe the design goals that our machine-code analysis framework must aspire to in order to be practical. Section 3.2 gives an overview of our x86 ISA model, and describes how it can be used to state properties of programs. Section 3.3 discusses how our x86 ISA model can be used to verify program properties.

3.1 Design Goals of Analysis Framework

An objective of this research is to illustrate that machine-code verification is a viable course to ensure software reliability. In order to be effective and practical, our analysis framework must be developed with the following design goals in mind.

Accuracy Our x86 specification should accurately model the x86 ISA. Simplifications or omissions in the semantics of the ISA could lead to unreliable program analyses, thereby defeating the purpose of this undertaking. We ought to be capable of validating our x86 ISA model by performing co-simulations against a target system, which can either be a real x86 processor or a mainstream simulator like QEMU. That is, we should be able to execute a test program on the target system as well as on our model, and to compare their states after the execution of every instruction in the program. Performing co-simulations will make it possible to find and fix discrepancies in our x86 model, thereby

increasing trust in its accuracy.

Execution Efficiency Our x86 ISA model should support efficient execution of machine-code programs. A benefit of supporting efficient execution is that model validation via co-simulations will become practical. Another benefit of supporting efficient execution is that it will allow testing a machine-code program with different kinds of inputs in a safe and easily programmable environment. It is advisable to test a program so that its behavior and/or purpose can be understood before investing in effort-intensive formal verification. In this sense, our x86 model should provide capabilities similar to those of mainstream instruction-set simulators.

Reasoning Efficiency Our analysis framework should support efficient reasoning about machine-code programs in order to reduce user effort. To this end, reasoning must be automated as much as possible. Moreover, given the undecidable nature of program verification, the framework should offer ways to debug failed proof attempts. Duplication of user effort must also be avoided — our framework should be equipped with general-purpose libraries that can be re-used for the analysis of various kinds of programs.

Usability Balancing *verification effort* and *verification utility* is a highly pertinent issue. Depending on the rigor of analysis desired, the framework should provide different modes of operation. For example, when analyzing an application program, a user may want to operate at the same level of abstraction as that offered by an operating system when developing that program, i.e., where the x86 system state (which includes data structures like page tables) is hidden.

The user may assume the correctness or delay the analysis of the relevant services supplied by the operating system. Of course, when analyzing a system program, the user would require access to the entire x86 state. Another capability that the framework should include vis-à-vis usability is to allow the effective utilization of the x86 model as an instruction-set simulator. This includes providing tools that enable dynamic instrumentation of machine-code programs to monitor their behavior, and executable file readers and loaders to automatically parse machine-code programs and load them in the appropriate region of the model's memory.

These goals place inherently opposing demands on our analysis framework. For example, there is a trade-off between execution and reasoning efficiency. Specification functions used for reasoning are typically simple and easy to comprehend, and necessarily so, because they are trusted to be correct. However, these same functions will be used for execution. Optimizing these functions for execution efficiency can increase the complexity of their algorithms, making them difficult to understand and reason with. There is a tension between the goals of accuracy and usability as well. Adding new features and capabilities to our framework will aid in program analysis, but they will also increase our framework's complexity. This will raise questions about the accuracy of the x86 ISA model.

Features provided by the ACL2 theorem-proving system made it possible to attain all of these goals; these features are described in Chapter 4. A constant refrain across this dissertation is how we avoided compromising on our design goals by mitigating trade-offs.

3.2 Developing the x86 ISA Model

Due to the size and complexity of the x86 instruction-set architecture, the development of our x86 model was a demand-driven process. We began by supporting only those ISA features and instructions that were used by a program under scrutiny. Over time, our model evolved to contain a specification of a considerable portion of the IA-32e mode of the x86 ISA:

- The entire basic execution environment (which is defined by Intel as "a set of resources for executing instructions and for storing code, data, and state information"),
- All the system-level registers and data structures (e.g., those that support memory management via paging and segmentation),
- All the addressing modes for fetching operands of an instruction, and
- 413 x86 instruction opcodes, including arithmetic, floating-point, and control-flow instructions, and system-mode instructions like those for loading memory-management registers.

Some features that have been excluded from our x86 ISA model are I/O capabilities (e.g., `IN`, `OUT` instructions), interrupts and exceptions, task management facilities, multiple-processor management, performance monitoring and processor tracing mechanisms, caches and translation look-aside buffers (TLBs), power management, and virtual machine extensions. A reason why these features are not yet specified is because we did not reason about programs that interacted with these features at

this time. Another reason is that features like caches are mostly transparent to the programmer. Of course, the x86 ISA model can be extended to add support for these features — for some of these features (such as asynchronous events like interrupts, operation of multiple processors, etc.), this will be a formidable long-term project, whereas for the others (e.g., exceptions), this will be a relatively shorter-term project.

Our x86 model can be used to analyze *unmodified* programs emitted by mainstream compilers like GCC [32] and LLVM [47]. Our model is more than 40,000 lines of ACL2, excluding many blank lines and comments. We use the ability of ACL2/Lisp to treat code as data in order to automatically generate many functions and theorems about our specification functions. More details about the x86 ISA model are in Part II of this dissertation.

Our model in its current form was obtained after six revisions; twice, we started development completely from scratch. Every revision took months of dedicated effort and was subjected to code reviews and co-simulations for validation. Each revision overcame design shortcomings in the previous revision. For example, the first model enabled elegant reasoning but offered poor execution performance. The second model was re-written to provide the best execution performance possible but it made reasoning awkward. The third model improved the second one by using sophisticated abstraction techniques to attain both reasoning and execution efficiency. The fourth model was re-written to simplify bit-vector reasoning. The fifth model separated instruction decoding from instruction semantics, making the code base maintainable and easily extensible. The sixth (and current) model made modifications to the memory management specification so that a uniform memory

interface was available during reasoning.

How do we use our x86 ISA model to characterize a program's behavior? A program property is a statement about its behavior, and this statement is expressed in terms of states of computation — it can either be a characteristic of just one state (or a set of states) or a relationship between a set of final states and the initial states [72]. The computation done by a program can be described by how each of its commands or instructions transform a state to another [112, 159]. Thus, a program property can be described by defining the notion of a state of computation, capturing the effects of each instruction on this state, and finally, making a statement about a set of states.

We represent the behavior of x86 programs by constructing our model of the x86 ISA in an *interpreter-style of operational semantics* [102] using the ACL2 theorem-proving system. In this modeling style, an interpreter defined over the processor's state is used to ascribe meaning to programs. Typical of such a model, our x86 ISA specification has four main components.

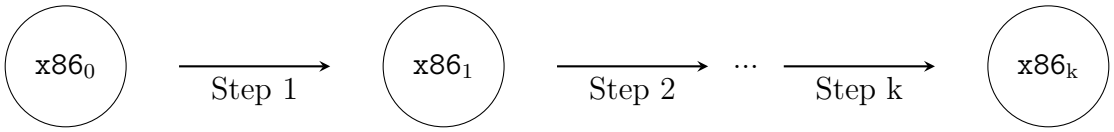

Figure 3.1: A Run of the x86 Interpreter that Executes k Instructions

1. **State**: The state consists of the components of the x86 ISA, like registers, memory, and flags.

2. **Instruction Semantic Functions**: An instruction semantic function defines the behavior of an x86 instruction. It takes an initial x86 state as input and returns a modified next state as output.

3. **Step Function**: A step function fetches an x86 instruction from the memory in the state, decodes it, and then executes it by dispatching control to the appropriate instruction semantic function.

4. **Run Function**: A run function takes the number `n` of instructions to be executed and an initial x86 state. It either takes `n` steps or terminates after taking `k` steps (`k < n`) if an unrecoverable error is encountered during the `(k+1)`$^{\text{st}}$ step. It returns an appropriately modified final x86 state in either case. See Figure 3.1 for an illustration of a run of an x86 machine-code program.

ACL2 is a first-order logic of recursive functions, so we can express a wide range of properties about x86 programs. One can write simple specification functions to convey a program's intent and connect them to the relevant components of the x86 state(s). For example, consider a sub-routine that computes the Fibonacci number corresponding to a given input `n`. The specification of this sub-routine is a simple recurrence relation that describes Fibonacci mathematically — that is, for $n \in \mathbb{N}$, $F(n) = F(n-1) + F(n-2)$ and $F(0) = 0$, $F(1) = 1$. Let this sub-routine execute from an initial x86 state where the input `n` is in register `rdi`. At the end of this sub-routine's execution, it produces a final x86 state where some component, say the register `rax`, is intended to contain the result. Then, the functional correctness of

this sub-routine[2] can be expressed by stating the following property, where i denotes the number of instructions to be executed for the sub-routine to run to completion:

$$\forall n,\ n \in \mathbb{N}:$$
$$n = \mathit{read\text{-}rdi}(x86) \wedge \mathit{read\text{-}rax}(\mathit{run}(i, x86)) = F(n)$$

Another example of a useful property is that a program never modifies a certain region of memory, represented by a set of addresses $addrs$, in any relevant set of executions, where l denotes the upper limit on the number of instructions to be executed. One way to state this property is as follows.

$$\forall i,\ i \geqslant 0\ \mathit{and}\ i \leqslant l:$$
$$\mathit{read\text{-}mem}(addrs, \mathit{run}(i, x86)) = \mathit{read\text{-}mem}(addrs, x86)$$

Note that these properties (like other program properties in our framework) are expressed in terms of our x86 interpreter (i.e., its run function).

3.3 Verifying Properties of x86 Machine Code

The behavior of a program can be described by composing the effects of each constituent instruction on the x86 state. Broadly speaking, each x86 instruction can be described in terms of two operations: reads from and writes to the x86 state. For example, consider the `ADD` instruction (opcode 0x00) whose first operand is in either the memory or a general-purpose register and the second operand is in a

[2]For presentation purposes, we ignore the fact that the register widths are finite — `rdi` and `rax` can store up to 64-bit numbers. However, we do account for such issues in this research.

general-purpose register; the output is written to the location of the first operand. This instruction can be described as a series of the following operations (details like exceptions are excluded here):

- **Read** or fetch the instruction from the memory.
- **Read** the operands from the memory and/or general-purpose register(s).
- **Write** the computed sum to the memory or general-purpose register.
- **Write** the newly computed flag values to the flags register.
- **Write** the address of the next instruction to the instruction pointer register.

In order to verify a program property, we need to prove that it holds for all relevant executions of the program. Given a model defined using operational semantics, various proof strategies can be applied for program analysis, such as the clock functions approach and stepwise invariants approach [156, 162, 163]. Central to all these approaches is the idea of *symbolically simulating* a program [105], i.e., executing it with symbolic values instead of concrete values, which allows capturing all possible program executions. A relevant set of executions can then be obtained by constraining the symbolic values appropriately. For example, if the user wished to characterize the behavior of a program for all unsigned integer inputs of magnitude 32 bits, even if the program could input arbitrary 64-bit signed integers, then symbolic values representing 32-bit unsigned integers can be used in order to prune away irrelevant executions.

Unlike concrete execution, which is natively supported by executable ACL2 functions, support for symbolic simulation needs to be built on top of our x86 model. Our framework consists of theorems, stored as ACL2 rewrite rules, that help in controlling symbolic simulation by describing the effects of read and write operations with symbolic values. We present four main kinds of these theorems below.

1. **Read-over-Write Theorems**: There are two types of Read-Over-Write theorems. The first describes the independence or non-interference of different components of the x86 state. For example, an update made to a specific register does not modify the value of the flags (or any other component) in the x86 state. The second kind of Read-over-Write theorem states that reading a component after a value `v` was written to it returns `v`, i.e., the old value is over-written.

2. **Write-over-Write Theorems**: There are two types of Write-over-Write theorems. The first asserts that independent writes to the x86 state can commute safely. The second says that the most recent write is the only visible write if consecutive writes are made to an x86 component, i.e., it overwrites the values of all previous write operations.

3. **Writing-the-Read Theorems**: Writing-the-Read theorems assert that if the value read from a component is written back to the same component, the resulting state is indistinguishable from the initial one, i.e., the effect of the write is immaterial.

4. **State Well-Formedness Theorems**: These theorems assert that writing a valid value to a component in a well-formed x86 state returns a new x86 state that is still well-formed.

Note that not only do these theorems pertain to *every* program supported by our x86 model, they describe the properties of the x86 ISA itself. Some examples of these theorems included in our analysis framework can be found in Appendix D.

Though obvious, these theorems are central to performing symbolic simulation. They allow expressing a state $\texttt{x86}_\text{f}$ `:= run(n, x86`$_\text{i}$`)`[3] as a nest of updates made to the starting state, $\texttt{x86}_\text{i}$. All updates that are overwritten by later updates are not reflected in $\texttt{x86}_\text{f}$'s representation. This representation is lean and readable; any improvement in readability is desirable because of the inherently low-level nature of machine-code analysis. These theorems, specifically the Read-over-Writes, also enable projecting out components from $\texttt{x86}_\text{f}$ that are relevant to the property under consideration.

The heuristics of the ACL2 theorem prover are influenced by the database of known theorems/rules, and developing lemma libraries with efficient rules is critical to automate program verification. These lemmas need to be as general as possible in order to facilitate their re-use. Our lemma libraries for user-mode and supervisor-mode program verification, are around 16,000 lines of ACL2, excluding comments and blank lines.

[3]Note that `n`, the number of instructions to be executed, can be symbolic too.

More details about controlling the symbolic simulation and proving properties of x86 programs using our analysis framework are presented in Parts III and IV respectively.

Chapter 4

ACL2 Theorem-Proving System

Over the years, well-known undertakings, some of which have been discussed previously in Chapter 2, have already demonstrated that theorem proving techniques can scale well and can be used to obtain various kinds of guarantees about computing systems. A couple of illustrative examples are the development and verification of the CLI stack [188] using NQTHM, a Boyer-Moore theorem prover, and the verification of the seL4 project [45, 92] using the Isabelle/HOL theorem prover.

Our theorem prover of choice is the ACL2 system, a descendant of the Boyer-Moore theorem prover. ACL2 (or ACL^2) stands for "A Computational Logic for Applicative Common Lisp". It is a first-order logic of recursive functions and a mechanical theorem prover used to prove theorems in that logic. Since it is based on an applicative subset of Common Lisp, it is also a programming language, which offers the execution efficiency provided by underlying Lisp compilers. ACL2 has proved its mettle as an industrial-strength system, and is regularly used in both academic and commercial applications [20, 183].

Unlike some higher-order theorem proving systems that require the extraction of executable definitions from formal specifications in order to perform co-simulations (and thus, also require trusting or proving that this extraction process is correct),

ACL2's logic is directly executable[1]. This permits building a unified model for program analysis — for both testing via concrete executions and formal verification via symbolic simulation.

ACL2 comes with many automated proof strategies, with conditional rewriting being its main workhorse. These strategies recursively decompose a given conjecture or goal until all its subgoals are proved. Users can provide hints to the prover to guide the proof attempt, and can even add their own proof strategies. ACL2 can also take advantage of the strengths of external deduction tools like SAT/SMT solvers [133]. A user interacts with ACL2 via a REPL (read-eval-print loop), and can store his functions and theorems in ACL2 libraries called *books*. ACL2 and its books are freely available online [18], and they include extensive documentation (over 23,000 topics [10, 86]). Books can be *certified* by ACL2 to ensure their soundness, and they can be *included* in ACL2 projects to build on existing definitions and theorems. In our project, we use several pre-existing ACL2 books, such as the `STD` books [7] for basic list processing theorems and for convenient macros for defining ACL2 events, the `ARITHMETIC` book [1] for basic arithmetic reasoning, the `BITOPS` books [8] for bit-vector reasoning, the `GL` books [4] for proving theorems about finite objects using bit-blasting, the `RTL` books [6] for specifying x86 floating-point instructions, and other miscellaneous books [5] to provide increased automation for certain kinds of ACL2 proofs.

Every modeling decision made over the course of this dissertation project was

[1]We can choose to use non-executable definitions in ACL2, but by default, definitions are executable.

guided by our design goals, discussed previously in Section 3.1. Our design goals dictate that our x86 model be optimized for both reasoning and execution efficiency. This places opposing demands on our unified model. Functions optimized for execution efficiency often follow a sufficiently different algorithm from the naïve one that their behavior is not obvious, which makes them poor candidates for specification functions. An illustrative example is that of the factorial function — the naïve algorithm uses recursion, but a practical one can use complicated techniques involving prime factorization [154]. ACL2 provides features, like `mbe` (discussed later in this chapter) and abstract stobjs (discussed in Chapter 5), that mitigate this trade-off between reasoning and execution efficiency.

We discuss some technical aspects of ACL2 to familiarize the reader with some ACL2-specific references made in this dissertation. ACL2, being an applicative subset of Lisp, uses its syntax — it has a parenthetical prefix notation. ACL2 comments begin with a semi-colon. The Lisp/ACL2 keyword `defun` is used to define a function. Functions may return multiple values in ACL2 by using the `mv` construct. For example, the function `id` below simply returns all three of its inputs.

```
(defun id (x y z)
  (mv x y z))
```

The form `mv-let` can be used to call multiple-valued functions. The following form binds the return values of the function `id`, called using arguments 1, 2 and 3, to local variables `x`, `y`, and `z`. The body of the `mv-let` contains the `let*` macro, which increments the return values of `id`, and binds them to its own local variables which are also called `x`, `y`, and `z`. It finally returns the list `(2 3 4)`.

```
(mv-let (x y z)              ;; local variables
        (id 1 2 3)           ;; multi-valued expression
        (let* ((x (+ 1 x))   ;; body
               (y (+ 1 y))
               (z (+ 1 z)))
          (list x y z)))
```

Such expressions are fairly common in our dissertation project, but for readability, we use the `b*` macro [2] to specify such computations. This macro is available as an ACL2 book.

```
(b* (((mv x y z) (id 1 2 3))
     (x (+ 1 x))
     (y (+ 1 y))
     (z (+ 1 z)))
  (list x y z))
```

Another notable macro is the Lisp built-in `cond`. This macro evaluates one test form at a time in order until a test form is found that evaluates to a non-`nil` value; it then evaluates the result form associated with that test form — the value returned by this result form becomes the return value of the entire `cond` expression. If no test form evaluates to a non-`nil` value, `cond` returns `nil`.

```
(cond ((<test-form-1> <result-form-1>)
       (<test-form-2> <result-form-2>)
       (<test-form-3> <result-form-3>)
       ...
       (<test-form-n> <result-form-n>)))
```

The ACL2 event `defthm` takes a conjecture as input and is used to name, prove, and store a theorem. User guidance can be given to the prover in a `defthm`

event via the keyword `:hints`. A `defthm` event also takes a keyword `:rule-classes`, which allows a user to specify which rules are to be built from the theorem. For example, the `:rewrite` rule class is used to build (possibly conditional) rewrite rules, and the `:congruence` rule class, discussed later in Chapter 10, is used to build rules that specify which equivalence relations are preserved when certain arguments of functions are being simplified. Similarly, `thm` is also used to prove a theorem and it takes user guidance in the same manner as `defthm`. However, it does not name the proved theorem or store it in the ACL2 database.

Below, we discuss some features of ACL2 used extensively in our project; more advanced features, like abstract stobjs, are discussed later in this dissertation.

Guards Before we can describe ACL2's guards mechanism, Lisp's notion of *intended domains* must be introduced. The Common Lisp standard [26] specifies an intended domain for each Lisp primitive function in which the return value of that function is defined. The behavior of a primitive outside this domain depends on the Lisp implementation. For example, the return value of the function `car` is specified only when its input argument is either `nil` or a `cons` object (i.e., an ordered pair) — in the former case, `car` returns `nil` and in the latter case, it returns the first component of the `cons`. The behavior of `car` for other types of inputs, say strings, can vary among Lisp implementations. Thus, Lisp functions are partial — they are logically well-defined only for a subset of all possible types of values.

However, all ACL2 functions are total because ACL2's *completion axioms* define the behavior of every primitive function for every possible input. The completion axiom of `car` is given by the following formula, which says that if `x` is outside the

intended domain of `car`, `car` returns `nil` in ACL2, irrespective of the underlying Lisp implementation. Note that the unary function `consp` is a *recognizer* of a `cons` object — that is, `consp` returns `t` when its input is an ordered pair and `nil` otherwise. It is customary to use `p` as a suffix of recognizer functions.

```
;; Completion axiom of car
(equal (car x)
       (cond ((consp x) ;; Test   1: Is x a cons pair?
              (car x))  ;; Result 1: Return first component of x
             (t         ;; Test   2: Otherwise...
              nil)))    ;; Result 2: Return nil
```

The ACL2 *guard* mechanism [13] is used to explicitly specify the intended domain of a function. For instance, the guard of `(car x)` is `(or (consp x) (equal x nil))`. When a function is *guard-verified* in ACL2, it means that that function respects the guards of all functions it calls. This allows a guard-verified ACL2 function to be executed safely in the host Lisp because the behavior of such a function will be consistent across all Lisp implementations. A guard-verified function is executed efficiently in the host Lisp as opposed to being evaluated potentially slowly in ACL2. Thus, guard verification plays a crucial role in improving the execution speed of a function. Guards have no effect during reasoning — they do not affect the definitional axiom that is added when an ACL2 function is admitted.

To summarize, ACL2 evaluates a call of a function exactly as Common Lisp does, except that it first uses guards to check if each function call is legal. For instance, executing `(car 42)` in ACL2 will result in a *guard violation*, but we can prove that `(car 42)` is equal to `nil`, thanks to its completion axiom.

Must Be Equal Another feature that can allow faster execution of ACL2 functions is Must Be Equal or `mbe` [96]. This feature allows the following form to be used in functions:

```
(mbe :logic <logic-code> :exec <exec-code>)
```

The `:logic` part can contain simple code amenable to reasoning and the `:exec` part can contain code optimized for execution efficiency. The `mbe` form generates proof obligations that are added to the proof obligations for guard verification. When these obligations are proved, `<logic-code>` and `<exec-code>` are established to be equal. Thus, during reasoning, an `mbe` form is equal to `<logic-code>`, and during execution, it is equal to `<exec-code>` when the function's guards are verified.

Type Declarations Common Lisp supports arbitrary-precision arithmetic [27]. Operations on *fixnums*, or machine integers, are fast because they are supported by built-in functions (basically, primitive machine instructions). Operations on *bignums*, which are Lisp integers that are stored using an "unlimited" number of bits, are slower because they are supported by a large arbitrary-precision library. The threshold for when an integer becomes a bignum is defined by the Common Lisp implementation[2]. A Lisp compiler uses arbitrary-precision arithmetic either when it is known that an operation involves bignums or if it cannot infer that the operation uses fixnums. Avoiding bignum operations when possible is important for execution efficiency. Our x86 model specifies the 64-bit mode of the x86 ISA, where many operations can

[2]CCL, our preferred Lisp implementation, defines 61 bit fixnums on 64-bit systems, i.e., they have a range from -2^{60} to $2^{60}-1$. Integers outside this range are bignums.

crossover to the bignum side. We describe our approach to optimize execution in such cases in the next chapter (specifically, Section 5.2.1).

ACL2/Lisp code can be annotated with type declarations, which convey type information to the underlying Lisp compiler so that it can generate efficient code. For example, a user might declare that a certain variable, say `x`, is always of type `(unsigned-byte 16)`, i.e., an unsigned integer of width 16 bits. Since 16-bit integers are fixnums on all modern Lisp implementations, the compiler is able to generate efficient machine code that does not involve either bignum computations or run-time type-checks involving `x`. Such type declarations must be justified in ACL2. They are added to the guard proof obligation to ensure that a function with type declarations can be safely executed in the host Lisp without the possibility of encountering any run-time type violations.

Part II

x86 ISA Model

Chapter 5

x86 State and Efficiency Concerns

The x86 ISA state is at the core of the processor's basic execution environment. It consists of a wide variety of registers, flags, and a huge memory. In principle, modeling the x86 state is straightforward. One can simply define a record data structure to specify the state as a collection of fields, each of which denotes an ISA component. There are two main requirements from our definition of the x86 state: one, the data structure should enable formal reasoning, and two, the same data structure should support efficient execution. For the first requirement, a formal data structure that provides the clean applicative semantics of *update-by-copy* may be used. For the second requirement, a fitting data structure would be one that provides efficient von Neumann semantics of *destructive updates*. However, these two usual solutions are inherently incompatible, thereby making the task of defining the x86 state for our framework non-trivial. The size of the memory provided by contemporary x86 implementations — 4096TB — poses another major challenge for modeling the x86 state. Clearly, defining the memory field akin to a simple array will require more RAM than is available in off-the-shelf computers.

We describe the x86 ISA components supported by our model in Section 5.1. In Section 5.2, we discuss how we define the x86 state such that it supports both

formal reasoning and efficient execution, paying special attention to the problem of modeling the memory. In Section 5.3, we present some implications for reasoning posed by the large number of fields in the x86 state's data structure, and discuss our solution that enables effective reasoning without compromising on efficient execution.

5.1 x86 ISA State Components

Table 5.1 lists the x86 components currently supported by our model.

Table 5.1: Components of the x86 ISA State

#	Component	Type and Size
1.	General-Purpose Registers	16 64-bit registers
2.	Instruction Pointer	1 64-bit register
3.	Flags Register	1 64-bit register
4.	Segment Registers	6 16-bit registers
5.	Segmented Memory Management Registers	2 80-bit registers
6.	Interrupt and Task Management Registers	2 16-bit registers
7.	Control Registers	16 64-bit registers
8.	Floating-Point Data Registers[a]	8 80-bit registers
9.	Floating-Point Control Register	1 16-bit register
10.	Floating-Point Status Register	1 16-bit register
11.	Floating-Point Tag Register	1 16-bit register
12.	Floating-Point Last Instruction Pointer	1 48-bit register
13.	Floating-Point Last Data Pointer	1 48-bit register
14.	Floating-Point Opcode	1 11-bit register
15.	XMM Registers	16 128-bit registers
16.	MXCSR Control and Status Register	1 32-bit register
17.	Machine-Specific Registers	6[b] 64-bit registers
18.	Byte-Addressable Main Memory	Models 2^{52} bytes

[a] MMX registers are aliased to the low 64 bits of the FPU's data registers, as dictated by the ISA.
[b] Intel defines a lot more than 6 MSRs. Our model currently supports 6 of them: `ia32_efer`, `ia32_fs_base`, `ia32_gs_base`, `ia32_kernel_gs_base`, `ia32_lstar`, `ia32_star`, `ia32_fmask`.

In addition to the x86 components presented in Table 5.1, our x86 state contains the following components that are an artifact of the model rather than the machine.

1. **Model State:** The `ms` field stores information about the status of the model. If the `ms` field is empty, then the x86 state is expected to reflect the real machine's state, and if not, then the run function terminates and execution is halted. Thus, an empty `ms` field obtained after a program's execution signifies that no model-related error was encountered *at any point* during execution. An example when `ms` is populated is when an unimplemented instruction is encountered.

2. **Fault:** The `fault` field is very similar to `ms` in the sense that a non-empty `fault` field will terminate the run function immediately. The difference lies in the intention — this field is populated when the processor itself would cause a fault that would be signaled by its exception-handling mechanism (e.g., page faults or general-purpose exceptions).

3. **Undefined:** Specifying the x86 ISA requires modeling undefined behavior as well. The `undef` field is instrumental in providing a pool of undefined values for use in instruction semantic functions. Implementation details are in Chapter 8.

4. **User-level Mode:** The field `user-level-mode` acts as a switch. When its value is `nil`, the x86 model is in the system-level mode; otherwise, it is in the user-level mode.

5. **Page-Structure Marking Mode:** The field `page-structure-marking-mode` acts as a switch for two sub-modes of the system-level mode. When its value is `nil`, then we are in the *system-level non-marking mode*, where the updates to accessed and dirty flags in the paging data structures are turned off. Otherwise, we are in the *system-level marking mode*, where these updates are turned on and the model is true to the real machine. Details are in Chapter 10.

6. **Environment:** The `env` field models an external environment for use in the user-level mode. This field includes a file system specification and an oracle. More details are in Chapter 8.

7. **Operating System:** System call implementations differ among different operating systems. This field, `os-info`, indicates which OS's system call service is being provided in the user-level mode. Details are in Chapter 8.

5.2 x86 ISA State Definition

We now discuss our definition of the x86 state that addresses the disparate requirements for execution as well as reasoning. First, we describe our original definition in Section 5.2.1 that provides efficient execution but falls short for effective reasoning. Then, in Section 5.2.2, we describe how we defined a "layer" on top of the original definition that provides a convenient interface for reasoning while preserving the efficiency benefits of the original definition.

5.2.1 Concrete Stobj Representation

We use an ACL2 data structure called *concrete stobj* [157] to model the x86 state. "Stobj" stands for "Single-Threaded OBJect", and it provides copy-on-write semantics for reasoning and destructive updates for efficient execution. Consider the ACL2 `defstobj` event below that introduces a stobj called `foo`.

```
(defstobj foo
  (field1 :type (array (signed-byte 64) (2)) :initially 0)
  (field2 :type (unsigned-byte 16)           :initially 10))
```

Logically, `foo` is a linear list of two elements; the first element `field1` is itself a linear list of two elements, each of which is a 64-bit signed integer with an initial value 0, and the second element `field2` is a 16-bit unsigned integer with an initial value 10. Under the hood, stobjs are implemented as Lisp vectors; `foo` is a vector with `field1` and `field2` as simple arrays of 2 and 1 elements, respectively. ACL2 sequences updates made to a stobj by enforcing some syntactic restrictions on its use. This ensures that only one instance of `foo` exists at any time, thereby providing high execution performance by facilitating destructive updates. In addition to introducing a data structure, a `defstobj` event also introduces some functions: *recognizers* that return `t` when the stobj and its fields have the right logical representation and `nil` otherwise, a *creator* that creates an initial logical representation of the stobj, *accessors* that read a stobj field, and *updaters* that write to a stobj field. For example, the recognizer, accessors, and updaters associated with the stobj `foo` are as follows. The guards of all these ACL2 functions have been elided here.

```
;; Recognizer for foo
(defun foop (foo)
  (and (true-listp foo)
       (= (length foo) 2)
       ;; field1p is the recognizer for field1.  It returns t if
       ;; the first component of foo is a list of 64-bit signed
       ;; integers, and nil otherwise.
       (field1p (nth 0 foo))
       (equal (len (nth 0 foo)) 2)
       ;; field2p is the recognizer for field2.  It returns t if
       ;; the second component of foo is a 16-bit unsigned
       ;; integer, and nil otherwise.
       (field2p (nth 1 foo))
       t))

;; Accessor for field1
(defun field1i (i foo)
  (nth i (nth 0 foo)))

;; Accessor for field2
(defun field2 (foo)
  (nth 1 foo))

;; Updater for field1
(defun !field1i (i v foo)
  (update-nth-array 0 i v foo))

;; Updater for field2
(defun !field2 (v foo)
  (update-nth 1 v foo))
```

The default names for the updater functions are `update-field1i` and `update-field2`, but we shorten them to `!field1i` and `!field2` respectively for convenience. We use this `!` notation to refer to updater functions throughout this dissertation.

The `defstobj` form that introduces the x86 concrete stobj is presented in Appendix A.1 — we use `x86$c` to refer to this object. The accessor and updater functions of the x86 concrete stobj are inlined for execution efficiency.

Execution Efficiency and Bignum Operations The choice of the types of fields in concrete stobjs can also affect execution performance. For example, if a field's

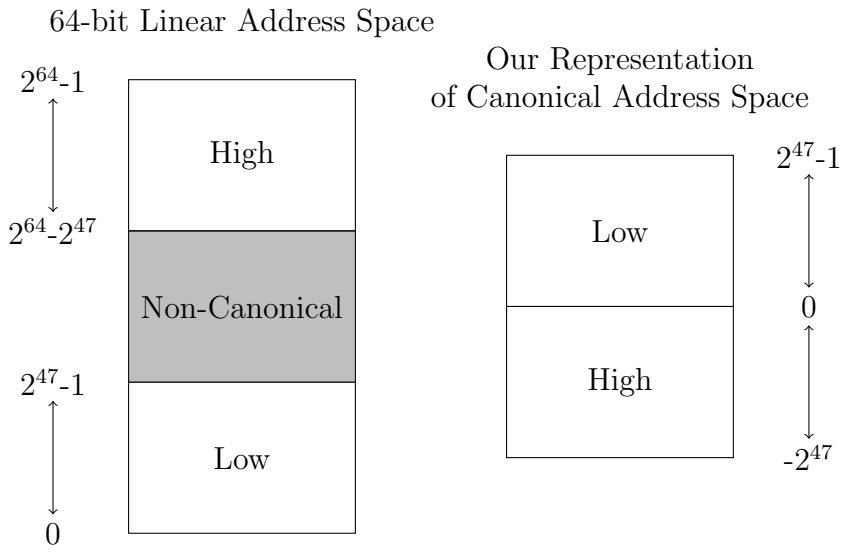

Figure 5.1: Representation of the 64-bit Canonical Address Space

type is such that it causes many bignum operations, the execution efficiency suffers significantly. Consider the 16 64-bit general-purpose registers (which include `rax`, `rbx`, etc.). One way to define the field corresponding to these registers is as an array of 64-bit unsigned integers. The problem with this choice is that large values in these registers would be stored as bignums. Instead, we define this field as an array of 64-bit *signed* integers, which allows large positive values to be stored as small negative ones; e.g., bignum 2^{64}-1 can be represented by the fixnum -1. Another example is that of the 64-bit instruction pointer `rip`. Like the general-purpose registers, we could have defined `rip` as a signed 64-bit integer; however, we define it as a signed 48-bit integer. This is because `rip` can contain only *canonical addresses*, which are defined in IA-32e mode as follows:

"A canonical address must have bits 63 through 48 set to zeros or ones (depending on whether bit 47 is a zero or one)" [38].

Thus, canonical addresses range from 0 to 2^{47}-1 and 2^{64}-2^{47} to 2^{64}-1, inclusive, in the 64-bit mode. In our model, 0 to 2^{47}-1 represents the lower range of canonical addresses and -2^{47} to -1 represents the upper range of canonical addresses, thereby ensuring that `rip` always contains a fixnum. See Figure 5.1 for an illustration. Similar optimizations have been made for other fields in the x86 state.

Memory Specification Note that Table 5.1 shows that our model supports a byte-addressable main memory of size 4096TB (2^{52} bytes). How do we define such a large object? Allocating 4096TB at once is impractical, if not impossible. One could imagine using a re-sizable array to model the main memory, but if a program were to read or write a sufficiently large address, say 0x10000000500, then the array would blow up to more than 1TB in size, which is already quite large. This memory footprint problem only gets worse for higher addresses.

To solve this problem, Hunt and Kaufmann implemented a time- and space-efficient memory model that is similar to how the x86 paging mechanism provides the illusion of a larger virtual memory than the available main memory [182]. Their work specified a quadword-addressable 256TB memory with 48-bit physical addresses, and we adapted it for this dissertation project to model a byte-addressable 4096TB memory with 52-bit physical addresses. Figure 5.2 illustrates how these three fields work together to specify the memory in our x86 ISA model using a memory write operation as an example. Memory is allocated on demand in blocks of size 128MB. Instead of a single field in the concrete stobj, three fields — `mem-table$c`, `mem-`

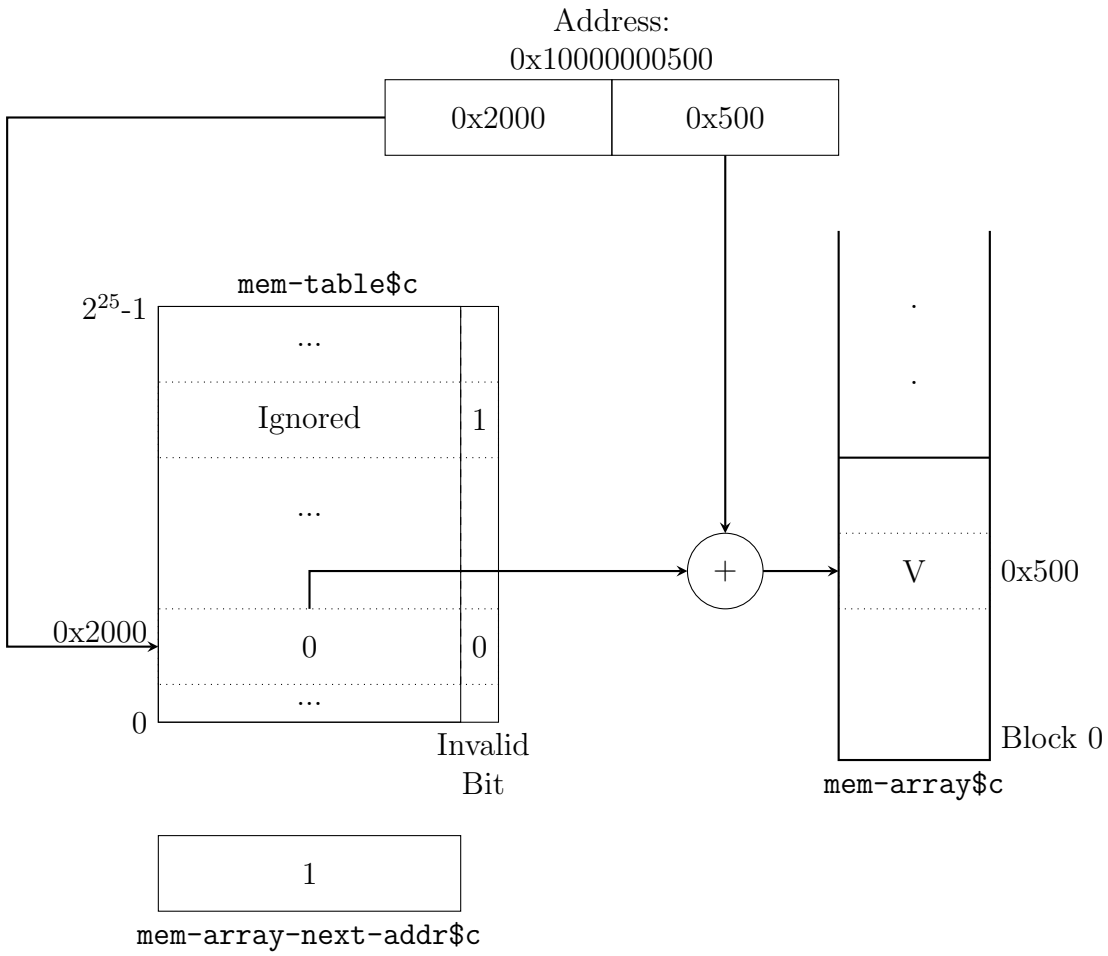

Figure 5.2: Fields Used to Specify the x86 Memory: This diagram shows how a write of value V to the physical address 0x10000000500 is accomplished in our memory model, assuming that we started with the initial state where `mem-array-next-addr$c` had value 0 and all entries in `mem-table$c` were invalid. The top 25 bits of this address form the value 0x2000, which is used to index into the `mem-table$c`. The initial value of `mem-array-next-addr$c`, 0, is written at this entry of the `mem-table$c`; it now points to block 0 of `mem-array$c`. Field `mem-array-next-addr$c` now contains the value 1. The low 27 bits of the address form the value 0x500, which is used as an offset within block 0 to locate the address where V is to be written.

Now, suppose the address 0x10000000501 needs to be accessed (for either a read or write operation). The same entry in `mem-table$c` will point to block 0, i.e., `mem-array$c` need not be re-sized. The offset into block 0 for this address will be 0x501.

`array$c`, and `mem-array-next-addr$c` — are used to specify the memory. Each entry in the first field `mem-table$c` can point to a 128MB block. This field, indexed by the top 25 bits of the physical address, is an array of 2^{25} elements, each of which is a 26-bit integer. The least significant bit of each entry is the *invalid bit*, which is 1 by default to indicate that the corresponding 128MB block is unallocated. The second field `mem-array$c` is the "real" memory that contains the bytes; it is indexed by the value obtained by the concatenation of the top 25 bits of the appropriate `mem-table$c` entry with the least significant 27 bits of the physical address. The field `mem-array$c` is re-sizable. The third field, `mem-array-next-addr$c`, is a 25-bit field that contains the address of a free 128MB block that will be allocated next, and its initial value is 0.

Consider the memory footprint of our model when writing a value at the address 0x10000000500, as illustrated in Figure 5.2. The memory allocated would be 128MB for a block of `mem-array$c` (which may increase if writes are made to other blocks), plus 104MB for `mem-table$c` (which remains constant, irrespective of other writes); compare this to the ∼1TB footprint with the naïve approach.

The following pseudocode describes the memory accessor function, `mem$ci`, where `x86$c` refers to the concrete stobj modeling the x86 state, `mem-table$ci` and `mem-array$ci` denote the accessor functions for the `mem-table$c` and `mem-array$c` fields respectively, and the notation `x[to:from]` represents the slice of `x` from bit position `to` to bit position `from`, inclusive of both indices.

```
mem$ci(physical-address, x86$c):
 mem-table-index := physical-address[51:27]
```

```
 mem-table-value := mem-table$ci(mem-table-index, x86$c)
 if (mem-table-value == 1) then
   // Block invalid or not present, which means no update was
   // done to the addresses in the 128MB block referenced
   // by mem-table-index.
   return 0
 else
   byte-index := physical-address[26:0]
   mem-array-index := concat(mem-table-value[25:1], byte-index)
   return mem-array$ci(mem-array-index, x86$c)
 endif
```

The following pseudocode describes the memory updater function `!mem$ci`, where `!mem-table$ci`, `!mem-array$ci`, and `!mem-array-next-addr$c` denote the updater functions for the `mem-table$c`, `mem-array$c`, and `mem-array-next-addr$c` fields respectively.

```
!mem$ci(physical-address, value, x86$c):
 mem-table-index := physical-address[51:27]
 if (mem-table$ci(mem-table-index, x86$c) == 1) then
   // Increase the size of mem-array$c field.
   new-block-addr := mem-array-next-addr$c(x86$c)
   new-mem-table-value := concat(new-block-addr, 0)
   x86$c := resize-mem-array$c(x86$c)
   x86$c := !mem-array-next-addr$c(1 + new-block-addr)
   x86$c := !mem-table$ci(mem-table-index, new-mem-table-value, x86$c)
 endif
 mem-table-value := mem-table$ci(mem-table-index, x86$c)
 byte-index := physical-address[26:0]
 mem-array-index := concat(mem-table-value[25:1], byte-index)
 x86$c := !mem-array$ci(mem-array-index, value, x86$c)
 return x86$c
```

We never access or update the three memory fields directly; instead, we always use `mem$ci` and `!mem$ci`.

5.2.1.1 Issues with the Concrete Stobj Representation

Though the use of concrete stobjs to represent the x86 state reduces the memory footprint of our x86 model, it presents the following two challenges.

Issue (1) Expensive guard checking: A well-formed x86 state is one that satisfies the predicate `x86$cp`, which is a conjunction of the stobj's native recognizer function as well as another predicate, `good-memp`, that states that the relationship among the three memory fields gives a correct model of a 4096TB byte-addressable memory.

```
(defun x86$cp (x86$c)
  (and (x86$cp-pre x86$c)
       (good-memp x86$c)))
```

If the `x86$c` stobj does not satisfy `x86$cp` (say, if `(good-memp x86$c)` evaluates to `nil`), then it means that it does not correspond to the x86 ISA state. Therefore, any function, say `f`, which takes the concrete x86 stobj as input would require `x86$cp` as a guard. Recall that guards play a crucial role in improving execution efficiency of an ACL2 function — see Chapter 4 for details. However, `good-memp` is an expensive predicate because it walks through each element of all the three memory-related fields. Whenever `f` is executed on concrete data in the ACL2 loop, costly guard checking adversely affects its execution efficiency.

Issue (2) Large logical representation of the x86 state: The logical representation of `mem-table$c` and `mem-array$c` is extremely large; the former is a linear list of 2^{25}

elements and the latter is a linear list that increases by 2^{27} elements at a time if more memory is requested. The size of these lists prohibits the use of bit-blasting tools to reason about programs (more about bit-blasting in Chapter 11); these lists would have to be created in order to bit-blast instructions that access and/or update memory.

A way to address issue (1) would be to turn off guard checking, which will prevent the execution of the expensive recognizer function, `good-memp`. However, this may actually slow down execution because functions will be evaluated in ACL2 as opposed to executed in the host Lisp. Moreover, no guard violations are reported when guard checking is off — thus, evaluating functions on inputs outside their intended domain might produce results that surprise and confuse us.

One could imagine resolving issue (2) by manually defining an alternative logical representation of the concrete state that has a sparse logical representation of memory (for example, association lists instead of linear lists); this new logical representation would be provably equivalent to the concrete stobj's representation. However, this manual solution would impose an enormous burden on the user when verifying program properties. We illustrate this point using a proof sketch about an arbitrary machine model below.

We start out with a concrete model, where the state is optimized for execution efficiency (e.g., the three memory fields in our concrete stobj) and the definitions have the suffix *$c*. We then decide to define an alternative version of the concrete model, called the *abstract* model, where the state is optimized for reasoning efficiency (e.g., a single memory field, such as a normalized association list [131]) and the definitions

have the suffix *$a*. We intend to use the abstract model for proofs and the concrete model for execution. Since both the abstract and concrete models should specify the same machine, we need to ascertain whether the theorems obtained using the abstract model are valid for the concrete model as well. After all, model validation will be done using the concrete model, and we can trust a theorem obtained using the abstract model only if it holds for this validated concrete model too.

Imagine that we have already proved *program-correct-abstract* below using the abstract model; *program-correct-abstract* says that if some pre-conditions hold over the abstract state, then some post-conditions hold over the final state obtained using the `run$a` function. The well-formedness of the x86 state (represented by predicates *x86$cp* and *x86$ap* for concrete and abstract states, respectively) is included in the preconditions.

Theorem: program-correct-abstract

$$preconditions\$a(x86\$a) \implies postconditions\$a(run\$a(n, x86\$a))$$

We now need to prove the following:

Conjecture: program-correct-concrete

$$preconditions\$c(x86\$c) \implies postconditions\$c(run\$c(n, x86\$c))$$

In order to prove *program-correct-concrete*, we would prefer to use *program-correct-abstract*; the alternative is to re-do this proof using `$c` functions, which defeats the purpose of defining an abstract model for ease of reasoning in the first place. To this end, we define a correspondence relation, say *corr*, between these two states. We

then prove *corr-init-states-implies-corr-final-states*, which asserts that if the initial concrete and abstract states correspond as dictated by *corr*, then the states obtained from the concrete and abstract run functions also correspond.

Theorem: corr-init-states-implies-corr-final-states

$corr(x86\$a, x86\$c) \implies corr(run\$a(n, x86\$a), run\$c(n, x86\$c))$

Note that the proof of *corr-init-states-implies-corr-final-states* is extremely tedious, if not very intellectually challenging; it involves proving the correspondence of the states returned by each instruction semantic function, which, in turn, requires proving the correspondence for each function that accesses and updates the states. In this same vein, we prove the two theorems below — the first, *corr-states-implies-preconditions*, says that if the states correspond and the concrete preconditions hold, the abstract preconditions also hold; and the second, *corr-states-implies-postconditions*, says that if the states correspond and the abstract postconditions hold, then the concrete postconditions also hold.

Theorem: corr-states-implies-preconditions

$corr(x86\$a, x86\$c) \wedge preconditions\$c(n, x86\$c) \implies$

$preconditions\$a(n, x86\$a)$

Theorem: corr-states-implies-postconditions

$corr(x86\$a, x86\$c) \wedge postconditions\$a(n, x86\$a) \implies$

$postconditions\$c(n, x86\$c)$

Note that the two theorems above can be made stronger in the following way by using equality instead of implication. However, we need not prove these theorems because their weaker versions above suffice for the rest of this exercise.

Theorem: corr-states-implies-equal-preconditions

$corr(x86\$a, x86\$c) \implies$

$preconditions\$a(n, x86\$a) = preconditions\$c(n, x86\$c)$

Theorem: corr-states-implies-equal-postconditions

$corr(x86\$a, x86\$c) \implies$

$postconditions\$a(n, x86\$a) = postconditions\$c(n, x86\$c)$

Given all these theorems, we are able to prove *program-correct-concrete-helper*.

Theorem: program-correct-concrete-helper

$corr(x86\$a, x86\$c) \wedge preconditions\$c(x86\$c) \implies$

$postconditions\$c(run\$c(n, x86\$c))$

We need to eliminate the first hypothesis of this theorem to obtain our target theorem *program-correct-concrete*. Thus, we define a function called *create-abstract-from-concrete-state* that takes a well-formed concrete state as input and creates an abstract state that corresponds to it. We also prove *create-abstract-from-concrete-state-and-corr* that assures us that this function is correct.

Theorem: create-abstract-from-concrete-state-and-corr

$x86\$cp(x86\$c) \implies corr(create\text{-}abstract\text{-}from\text{-}concrete\text{-}state(x86\$c), x86\$c)$

Finally, from *program-correct-concrete-helper* and *create-abstract-from-concrete-state-and-corr*, we can prove *program-correct-concrete*.

Clearly, this approach demands considerable effort from the user; it requires maintaining two models consistently. We need two functions — one each for the concrete and the abstract states — for every concept. For a model as large as the x86 ISA, this would make maintenance difficult. Also, we would have to prove the equality of values or the correspondence of x86 states returned by the concrete and abstract versions of every function that accesses and/or updates the state. Moreover, we would have to derive the program correctness proof on the concrete model from the corresponding proof on the abstract model for every program that we analyze.

Our research motivated the ACL2 developers to add a new feature to ACL2 called *abstract stobj* [168] that allows specifying an alternative logical representation (or an *abstract* representation) of a concrete stobj, in order to avoid compromising on three of our design goals — execution efficiency, reasoning efficiency, and usability; we helped in testing that new feature. Abstract stobjs reduce user overhead by making it possible to prove the correspondence between the concrete and abstract representations once and for all for a fixed set of functions, instead of for every function and for every program under scrutiny, as illustrated in the above proof sketch. Details about the use of abstract stobjs in our x86 model are in Section 5.2.2 below.

5.2.2 Abstract Stobj Representation

We can overcome both the issues with concrete stobjs — expensive guard checking and large logical representation of the x86 state — by using *abstract stobjs*. Similar to our naïve solution to Issue (2) in Section 5.2.1.1, an abstract stobj provides an alternative logical representation of a corresponding concrete stobj. Figure 5.3 illustrates the relationship between an abstract stobj and a concrete one. Let `x86` be an abstract stobj, `x86$c` a corresponding concrete stobj, and `f` be an *interface* function (also called a *native* function) associated with functions `f$a` and `f$c` that update the abstract and concrete stobj, respectively. Then, instance $\texttt{x86\$c}_1$ of `x86$c` corresponds to the instance $\texttt{x86}_1$ of `x86` if:

- `f$a` maps instance $\texttt{x86}_0$ of `x86` to $\texttt{x86}_1$.
- `f$c` maps instance $\texttt{x86\$c}_0$ of `x86$c` to $\texttt{x86\$c}_1$.
- The correspondence predicate holds for $\texttt{x86\$c}_0$ and $\texttt{x86}_0$.

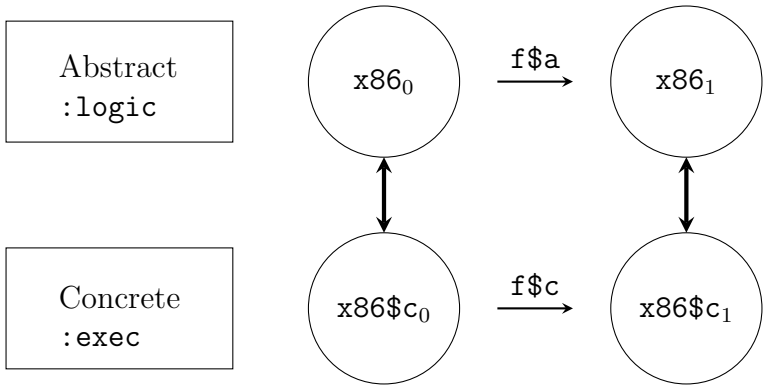

Figure 5.3: Corresponding Abstract and Concrete x86 States

In order to admit an abstract stobj, we need to provide the correspondence predicate and discharge the following kinds of proof obligations. These proof obligations are automatically generated by `defabsstobj`, the ACL2 event used to admit an abstract stobj; thus, the user does not have the burden of discovering the right kind or form of theorems needed to define an alternative representation of a concrete stobj.

1. **Correspondence Theorems:** These theorems state that the concrete and abstract states correspond at all times. There are three types of correspondence theorems:

   (a) *Creator Correspondence Theorem:* This theorem guarantees that the initial concrete and abstract stobjs (i.e., created via creator functions) correspond.

   (b) *Accessor Correspondence Theorems:* This type of theorem guarantees that an accessor function `f$a` for the abstract stobj and its counterpart `f$c` for the concrete stobj, produce the same value as output when applied to corresponding stobjs.

   (c) *Updater Correspondence Theorems:* This type of theorem guarantees that an updater function `g$a` for the abstract stobj and its and its counterpart `g$c` for the concrete stobj produce corresponding stobjs as output when given corresponding stobjs as input.

2. **Preservation Theorems:** These theorems guarantee that the recognizer always holds for the abstract stobj. There are two types of preservation theorems:

(a) *Creator Preservation Theorem:* This theorem guarantees that the creator function of the abstract stobj creates an object that satisfies the abstract stobj recognizer.

(b) *Updater Preservation Theorem:* This type of theorem guarantees that a well-guarded call of an updater function `g$a` outputs an object that satisfies the abstract stobj recognizer.

3. **Guard Theorems:** Guard theorems guarantee that the guards of the accessors and updaters `f` and `g` imply the guards of `f$c` and `g$c`, thereby ensuring that the concrete functions are called only when the guards of the native functions (and hence the guards of the concrete functions) are satisfied.

Once an abstract stobj has been admitted, any native accessor, say `f`, or updater, say `g`, reduces to functions `f$a` or `g$a` associated with the abstract stobj during reasoning and to functions `f$c` or `g$c` associated with the concrete stobj during execution. This means that there is a uniform interface to the state. Compare this to our naïve solution to Issue (2). With abstract stobjs, we only need to prove the correspondence, preservation, and guard theorems for each native function — we do not have to define two versions of every other function that operates on the state, and hence, there is no notion of two separate models. For example, all the instruction semantic functions are in terms of the native accessors and updaters, `f` and `g`. We do not need theorems like *corr-init-states-implies-corr-final-states*, *corr-states-implies-preconditions*, and *corr-states-implies-postconditions*. We only need to prove one version of a program's correctness theorem that would require the same

proof tactics as those for the proof of *program-correct-abstract*.

We define the x86 abstract stobj corresponding to the concrete x86 stobj by using *records* [132] to model all the concrete array fields. A record is a finite, normalized, association list that associates keys with non-default values. We present the ACL2 events related to introducing the x86 abstract stobj (including the correspondence function) in Appendix A.2. Here we focus on describing our memory representation; other fields, like general-purpose registers, for which the concrete field is a linear list (logically) and the abstract field is a corresponding record, are straightforward. Memory is represented by a single record in the abstract state, which corresponds to the three memory fields in the concrete stobj. Top-level functions `memi` and `!memi` invoke access and update operations on the memory record field when reasoning and invoke `mem$ci` and `!mem$ci` on the concrete stobj during execution. The abstract memory contains only the values that have been explicitly written to the memory. As opposed to large linear lists of zeros for the concrete memory fields, the initial representation of the memory record field is just 0, which represents that all unmodified memory locations have the default value 0. This results in a smaller x86 state that is more amenable to reasoning, thereby solving Issue (2). Aside: it is worth emphasizing here that even though an unmodified memory location — say, at physical address 100 — contains the default value 0 during execution, the following is *not* logically valid.

```
(implies (x86p x86) (equal (memi 100 x86) 0))
```

This is because the only fact known about `x86` above is that it satisfies its recognizer, `x86p`. The definition of `x86p` only includes the well-formedness of `x86`, which does

not say anything about its specific contents. Thus, unless we explicitly state the contents of `x86` during reasoning, we essentially account for all its possible values in our proofs.

Abstract stobjs also solve Issue (1), expensive guard checking, thanks to an optimization justified via preservation theorems. ACL2 does not generate certain proof obligations for guard verification because it is justified to assume that the abstract stobj recognizer always holds. Recall that there are three predicates: the native concrete stobj recognizer, the well-formed concrete x86 state recognizer (which is a conjunction of the native concrete stobj recognizer and `good-memp`), and the abstract stobj recognizer. The first predicate is known to hold for the concrete stobj for the purposes of guard verification and thus, it is not executed for guard checking. The second predicate was the cause of the expensive guard checking issue. However, analogous to the first predicate, the third predicate is known to hold for the abstract stobj because of the guarantees provided by the preservation theorems, and hence, it incurs no execution cost during guard checking.

A benefit of using abstract stobjs is that it allows optimizing the concrete state and functions for execution efficiency without affecting the abstract state and functions (and vice versa), as long as the correspondence relation is maintained. Thus, this successfully avoids trade-off between reasoning and execution efficiency.

5.3 Normalizing x86 State Accesses and Updates

Each component of the x86 state has an associated accessor and updater function. Previously, we had presented four main kinds of theorems about these

functions that are central to automating symbolic simulation of programs — the Read-over-Write, Write-over-Write, Writing-the-Read, and State Well-Formedness theorems. The total number of these theorems required is quadratic in the number of components of the x86 state [80]. Let n_{simple} be the number of non-array fields, n_{array} be the number of array fields, and $n = n_{simple} + n_{array}$ be the total number of components of the x86 state. The number of each kind of theorem required are as follows.

Read-over-Write n^2

Every accessor is paired with every updater.

Write-over-Write $n * (n - 1)/2 + n_{simple} + n_{array} + n_{array}$

The first term above is due to the pairing of every updater with every other updater. The second and third terms are due to the pairing of every updater with itself for consecutive (shadowed) writes to that field. The fourth term is due to the pairing of every updater of an array field with itself for commuting writes to distinct indices of the array.

Writing-the-Read n

Every updater is paired with its corresponding accessor.

State Well-Formedness n

Every updater needs to preserve the well-formedness of the x86 state.

The x86 state in our model has 28 components, with 16 simple fields and 12 array fields (see Appendix A for the definition of our x86 state), which means that

we need 1246 theorems. There are two problems here: one, adding new components to the x86 state to support an evolving model or ISA would entail proving more of these theorems, thereby increasing maintenance overhead; and two, a large number of theorems can slow down the theorem prover's rewriter.

An obvious solution to these problems is defining uniform accessor and updater functions, `xr` and `xw` respectively. These functions take field identifiers as input and call the corresponding native accessor or updater function. Now, the number of theorems required decreases significantly — we just have a small fixed number of theorems about these two functions to manage.

However, adopting this approach has an adverse effect on execution efficiency. Previously, inlined native functions were used to access or update the x86 state. With `xr` and `xw`, we incur the overhead of an extra function call. This might not seem like much, but considering that a single instruction would make several reads and writes via `xr` and `xw`, the overhead will add up over the course of execution of a program. Inlining `xr` and `xw` can avoid the cost of this extra function call, but because these functions contain a big `case` statement, this increases their code and memory footprint, which again impacts execution efficiency.

To solve this problem, we used ACL2's `mbe` feature to define new accessors and updaters for each component that serve as the top-level interface functions to the x86 state. The body of these new functions is an `mbe`, where the `:logic` part calls `xr` or `xw` and the `:exec` part calls the native accessor or updater. These new functions are kept enabled and inlined; the former implies that reasoning is done in terms of `xr` or `xw`, and the latter implies that during execution, the efficient native accessors and

updaters are called without incurring any function call overhead. We automatically generate these new functions from the definition of the x86 state. Thus, adding a new field to the x86 state will not require us to manually define the corresponding top-level accessor and updater function. See Appendix D for the ACL2 definitions of `xr` and `xw`, and the associated theorems that aid in symbolic simulation.

Chapter 6

Modes of Operation and Memory Interface

Our design goal of usability dictates that our framework should offer different modes of operation, as discussed previously in Section 3.1. To this end, our framework provides the *system-level mode* and *user-level mode*. The system-level mode is the true specification of the x86 ISA. The user-level mode is intended for the verification of application programs under the assumption that the underlying operating system services are correct. In this way, the user-level mode offers the same environment for analysis as is offered by an OS for program development. There are two key differences between these two modes of operation — the view of the memory they offer and the implementation of some instructions, with `syscall` and `sysret` being the most prominent ones. Apart from these differences, these two modes of operation share their code base, which is important from the point of view of maintainability and usability. In this chapter, we focus on the memory view and postpone the discussion of the difference in some instruction semantic functions to Chapter 7.

The x86 processors provide two main types of memories — *linear memory* and *physical memory*. Linear memory is an abstraction of the physical memory, and physical memory is the main memory addressed by the processor on its bus. Linear memory can be indexed either by a *logical address* or a *linear address*. When

using logical addresses, linear memory appears to consist of smaller regions called *segments*. *Segmentation* is used to map logical addresses to linear addresses. When using linear addresses, linear memory appears as a single large address space. *Paging* is used to translate a linear address to a *physical address*, which is used to index into the physical memory. See Figure 6.1 below for an illustration of the relationship between these addresses and the memories. Using paging, it is customary to define the mapping from linear to physical addresses in such a manner that a small physical address space can simulate a large linear address space; this larger linear address space is also referred to as *virtual memory address space*.

Figure 6.1: Types of Memory Addresses on x86 Machines

The system-level mode specifies physical memory (2^{52} bytes) and the user-level mode specifies linear memory (2^{64} bytes). The same memory field in the x86 state is configured to specify physical memory in the system-level mode and linear memory in the user-level mode. However, 64-bit programs, even those running in the supervisor mode, cannot access physical memory directly — linear memory is the only view of memory available to these programs. Therefore, the system-level mode contains the specification of IA-32e paging that maps linear memory to physical memory. Paging is unavailable in the user-level mode.

Clearly, the implementation of linear memory in the two modes of operation of our model is different. However, definitions of instruction semantic functions need

to use linear memory read and write operations in both the modes of operation. Thus, in order to enable code sharing between these modes, we need to provide a uniform linear memory interface. We define top-level functions, with *rm* and *wm* prefixes, to read and write linear memory for different units of data — `rm08` and `wm08` for bytes, `rm16` and `wm16` for words, `rm32` and `wm32` for doublewords, `rm64` and `wm64` for quadwords, and `rm128` and `wm128` for octawords. These functions simply call the appropriate user-level mode functions when the field `user-level-mode` in the x86 state is `t` and system-level mode functions otherwise.

We describe the memory specification in the system-level mode in Section 6.1, and then in the user-level mode in Section 6.2. In Section 6.3, we discuss how we normalize calls of all the linear memory accessor and updater functions in order to provide a uniform view of memory during reasoning.

6.1 System-level Mode

The system-level mode provides the same interface to programs as is provided by the x86 processor. The memory model in the system-level mode specifies 2^{52} bytes of physical memory, which is the largest physical address space provided by contemporary x86 implementations. Both the 64-bit x86 memory management mechanisms — paging and segmentation — are captured in their full detail in this mode.

Segmentation In the 64-bit mode, the base address of all segments, except those indicated by segment selectors `FS` and `GS`, is treated as 0. Certain checks like segment-limit checks and null-segment selector checks are not performed, but descriptor-table

limit checks, segment-type checks, and privilege-level checks are still enabled.

Figure 6.2 depicts how segmentation is used to obtain a linear address from a logical address in the 64-bit mode. A logical address consists of two parts — a *selector* and an *offset*. The visible part of the selector is 16-bits wide and it consists of three fields (not shown in Figure 6.2) — *requestor privilege level* (RPL), *table indicator*, and *index*. The index field is used to point to a data structure called a *segment descriptor*, which is located in a *descriptor table*. The descriptor table's base address is located in either the Global or Local Descriptor Table Registers (`GDTR` or `LDTR`). The table indicator denotes whether the descriptor table is global or local. The hidden part of the selector is populated with information present in the associated segment descriptors. A segment descriptor describes a segment — information contained in it includes the segment size, location, access control, and status information. The base address of the segment is obtained from the descriptor, and is added to the offset from the logical address to obtain the linear address. The RPL (along with other types of privilege levels like CPL, DPL, etc.) is used to check whether access is allowed to this segment or not.

Our framework models the segmentation process described above. Segmentation plays an important role in fast system calls. Unlike in the user-level mode, the `syscall` instruction's specification is exactly what is specified by the x86 ISA and the `sysret` instruction is available. These instructions involve changes in privilege levels and require access to system state, such as the machine-specific registers and code-segment descriptors, that describe where control is to be transferred (from application-level procedures to system-level procedures for `syscall` and back for

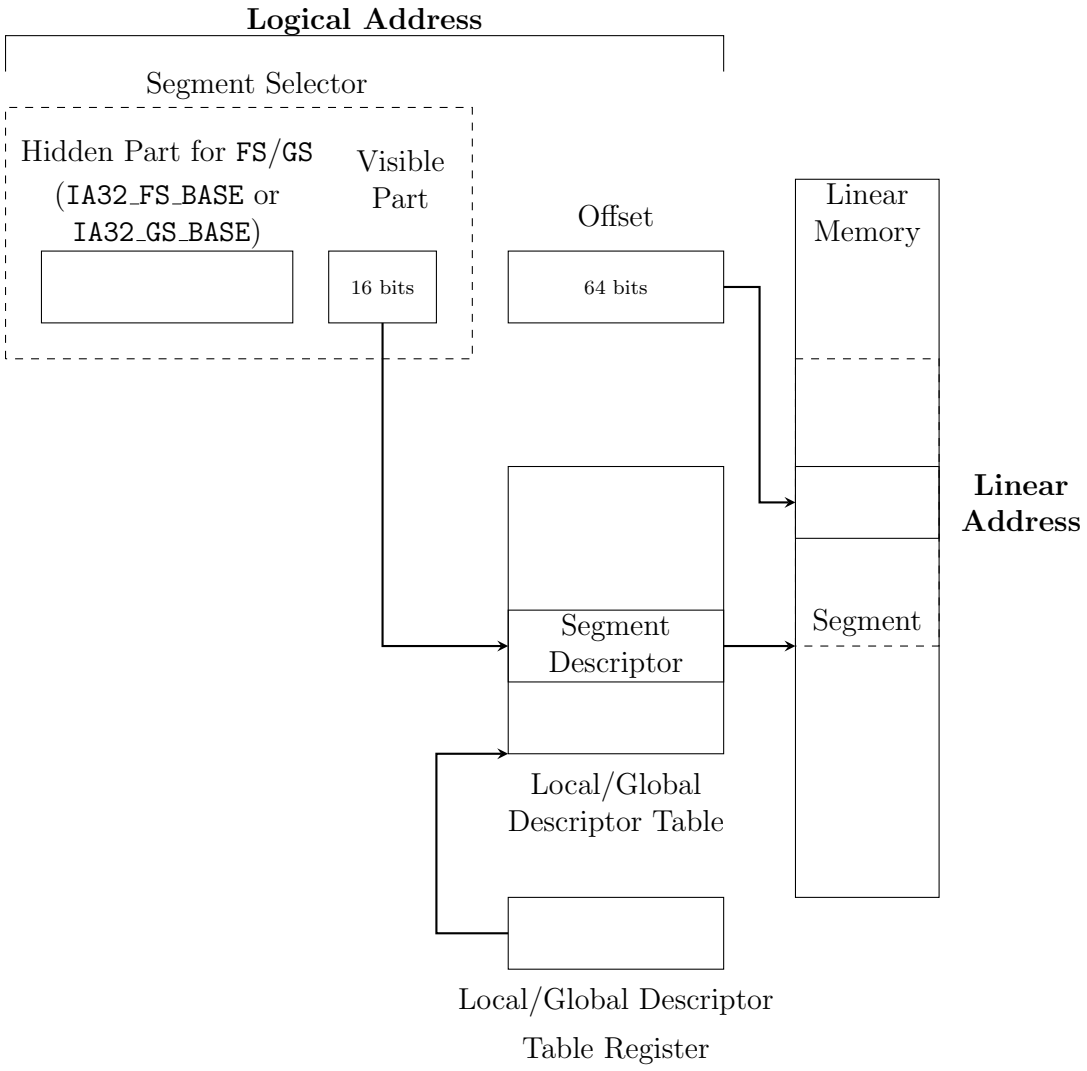

Figure 6.2: View of Segmentation in the System-level Mode

`sysret`) if all the privilege checks succeed.

Paging Paging is always enabled in the IA-32e mode, which is mode of operation of x86 processors added by Intel 64 architecture. In fact, according to Intel, "it is the use of IA-32e paging that defines IA-32e mode." Paging is responsible for linear-to-physical address translation and access rights management. Information about the map of linear to physical addresses, including access rights and cache types, is stored in hierarchical system data structures. The system-level mode supports all three configurations of these paging data structures — for 1GB, 2MB, and 4KB pages, as shown in Figures 6.3, 6.4, and 6.5 respectively.

We briefly describe how these paging data structures are used to perform address translation. Only canonical addresses are translated to physical addresses — the use of non-canonical addresses causes an exception (General Protection Exception, `#GP(0)`) on x86 machines. Recall that canonical addresses are linear addresses that have bits 63 through 48 set to zeros or ones (depending on whether bit 47 is a zero or one). In our model, canonical addresses are represented using a 48-bit signed integer, where 0 to 2^{47}-1 represents the lower range of canonical addresses and -2^{47} to -1 represents the upper range of canonical addresses. Bits 47-39 of a canonical address point to the `PML4TE`, an entry in the Page Map Level 4 Table whose base address is stored in the `cr3` control register. The next 9 bits point to the `PDPTE`, an entry in the Page Directory Pointer Table whose base address is located in the `PML4TE`. If the `PS` (page size) flag of `PDPTE` is set, then the `PDPTE` maps a 1GB page à la Figure 6.3, otherwise it provides the base address of another table called the Page Directory. The next 9 bits of the linear address index into the Page Directory

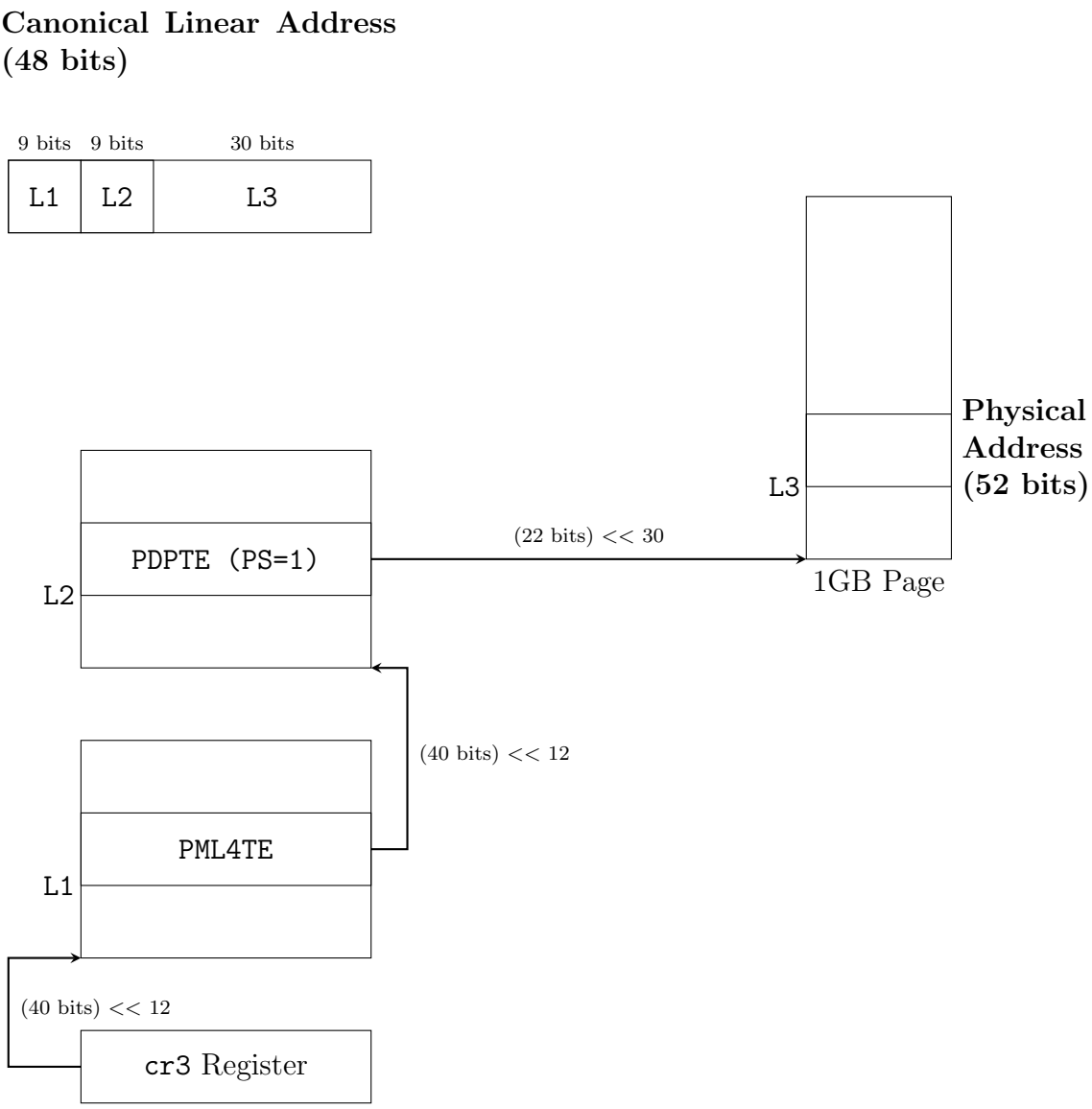

Figure 6.3: A Linear Address Mapped to a Physical Address in a 1GB Page

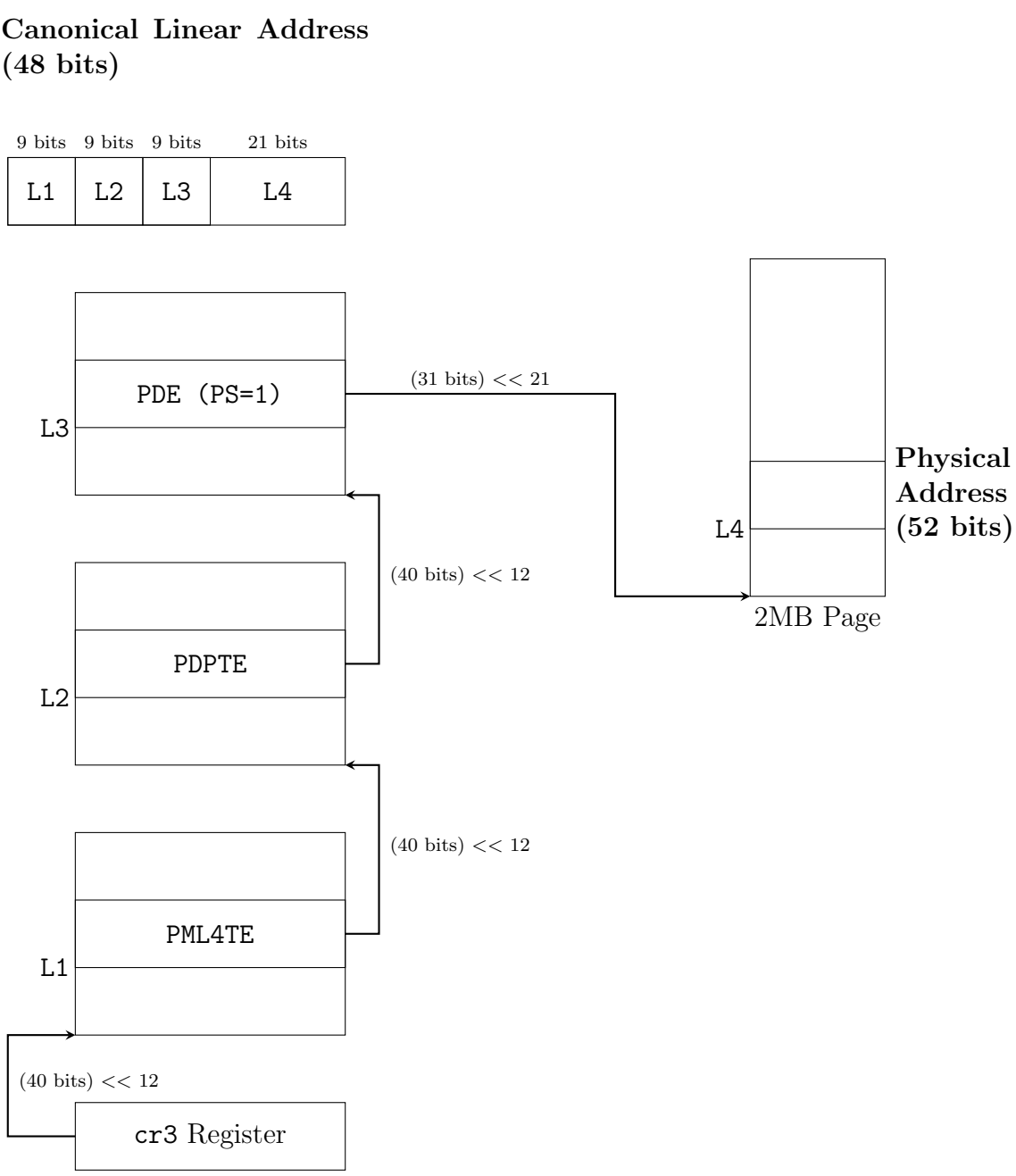

Figure 6.4: A Linear Address Mapped to a Physical Address in a 2MB Page

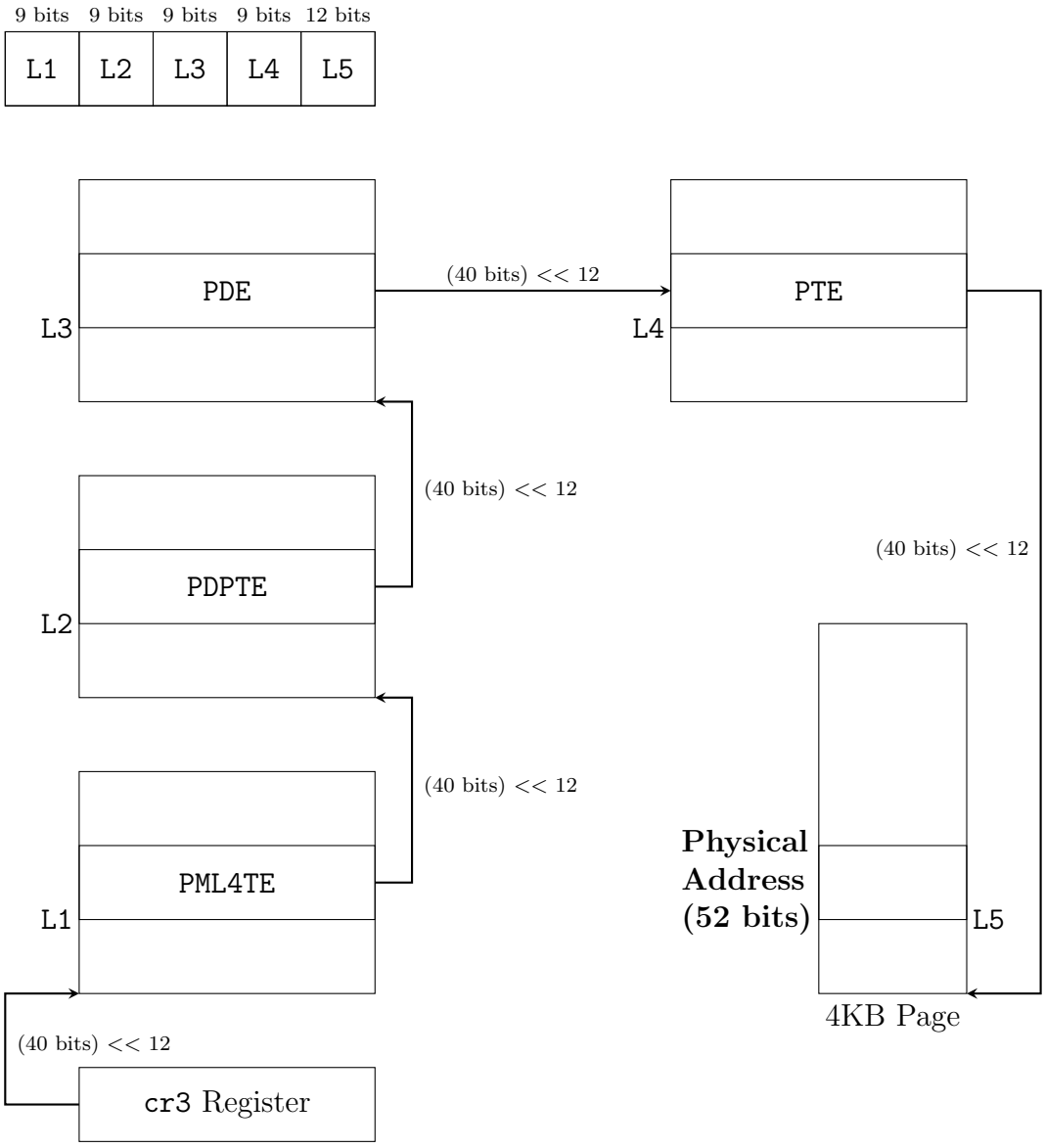

Figure 6.5: A Linear Address Mapped to a Physical Address in a 4KB Page

to point to the PDE. If the PS flag of PDE is set, then the PDE maps a 2MB page à la Figure 6.4, otherwise it references the Page Table. The rest of the bits in the linear address point to PTE, an entry in the Page Table, which finally maps a 4KB page, à la Figure 6.5. In this description above, we assume that the data structures are set up correctly and no page fault occurs — of course, our x86 model accounts for all such possibilities by setting the fault field in the x86 state when an exception is encountered.

We refer to the paging entries that are accessed during the translation of a linear address to its corresponding physical address as *translation-governing entries*. For example, a linear address mapped to a physical address located in a 4KB page has four translation-governing entries — PML4TE, PDPTE, PDE, and PTE.

The following pseudo-code denotes how linear memory is accessed and updated in the system-level mode.

```
rvm08-system-mode(lin-addr, permissions, x86):
  // Read a byte of virtual memory at address "lin-addr"
  // Returns an error flag (nil if no error), byte read,
  // and x86 state
 if non-canonical-address-p(lin-addr) then
    return (error, 0, x86)
 else
    // CPL is located in the RPL field of the CS segment register.
    cpl := get-rpl-from-cs-register(x86)
    [err, phyAddr, x86]:= la-to-pa(lin-addr, permissions, cpl, x86)
    if err then
       return (err, 0, x86)
    else
       byte := memi(phyAddr, x86)
       return (nil, byte, x86)
```

```
    endif
 endif

wvm08-system-mode(lin-addr, val, x86):
  // Write a byte "val" to virtual memory at address "lin-addr"
  // Returns an error flag (nil if no error) and x86 state
 if non-canonical-address-p(lin-addr) then
    return (error, 0, x86)
 else
    // CPL is located in the RPL field of the CS segment register.
    cpl := get-rpl-from-cs-register(x86)
    [err, phyAddr, x86]:= la-to-pa(lin-addr, permissions, cpl, x86)
    if err then
       return (err, x86)
    else
       x86 := !memi(phyAddr, val, x86)
       return (nil, x86)
    endif
 endif
```

The top-level linear memory functions `rm08` and `wm08` call these functions when the `user-level-mode` field is `nil`.

6.2 User-level Mode

The user-level mode provides the same interface to the x86 state as is provided by an operating system to application programs. The memory model in the user-level mode specifies linear memory, and thus, paging is unavailable here. However, this mode does provide support for *user-level* 64-bit segmentation.

Segmentation The Intel manuals say the following about 64-bit segmentation [37]:

In 64-bit mode, segmentation is generally (but not completely) disabled,

creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the effective address. The FS and GS segments are exceptions. These segment registers (which hold the segment base) can be used as additional base registers in linear address calculations. They facilitate addressing local data and certain operating system data structures.

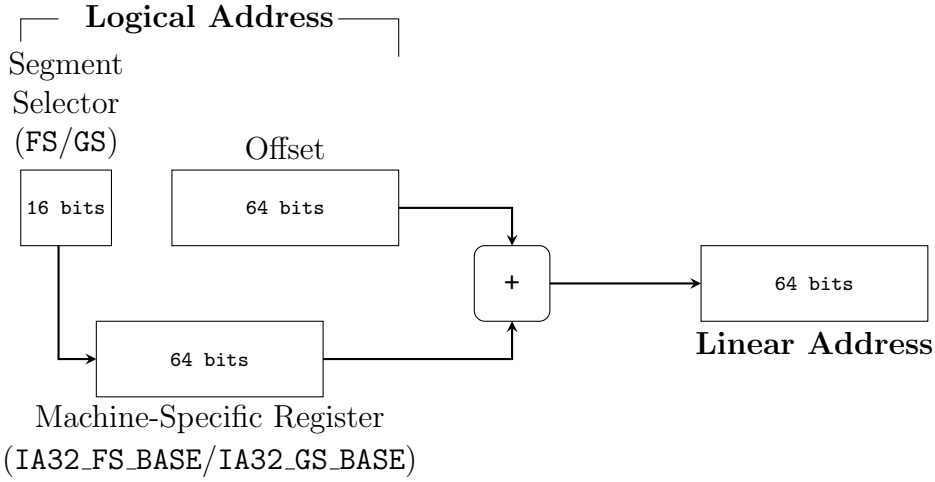

Figure 6.6: View of Segmentation in the User-level Mode

All the segment registers are associated with a hidden cache to store the corresponding information from the appropriate segment descriptor in the Global or Local Descriptor Tables (GDT and LDT). This cache is loaded automatically by the processor when the visible part (i.e., the 16-bit selector) is loaded. It is the responsibility of system software to load the visible part and, consequently, the hidden part, if the descriptor tables have been modified. In the user-level mode, it is assumed that the segment registers contain the appropriate values. Since system

software is assumed to be correct in the user-level mode, it is justified to make this assumption. Whenever a program uses logical addresses with `FS` or `GS` as segment selectors, the segment base addresses are obtained from the machine-specific registers `ia32_fs_base` or `ia32_gs_base`, respectively; see Figure 6.6. These registers are the hidden cache of the `FS` and `GS` segment registers.

Note that segmentation data structures, the GDT and LDTs, are unavailable in this mode, though they are specified in the system-level mode. Currently, application programs do not have a way of modifying segment selectors in the user-level mode. In order to do so, system calls can that enable such modifications to the system state can be implemented in the user-level mode — see Chapter 7, specifically Section 7.2, for information about support for system calls in our x86 ISA model.

Linear Address Space 64-bit application programs use 64-bit linear addresses. However, valid linear addresses have to be canonical (address ranges of 0 to 2^{47}-1 and 2^{64}-2^{47} to 2^{64}-1, which we prefer to represent using signed 48-bit integers in order to avoid bignum operations). This implies that the amount of linear memory *really* available to 64-bit programs is 2^{48} bytes, not 2^{64} bytes. Recall that the memory field in the x86 state specifies a byte-addressable memory with a capacity of 2^{52} bytes. We restrict the amount of memory available to programs in the user-level mode to 2^{48} bytes by defining linear memory accessor and updater functions, `rvm08-user-mode` and `wvm08-user-mode`, that require the address to be canonical, i.e., a 48-bit signed integer. In the definitions of these functions below, note that the address `addr` is converted to a 48-bit unsigned integer using the bit-vector operation `loghead` when indexing into our memory model because native functions `memi` and `!memi` take

unsigned integers as input.

```
rvm08-user-mode(addr, x86):
  // Read a byte of virtual memory at address "addr"
  // Returns an error flag (nil if no error), byte read,
  // and x86 state
 if canonical-address-p(addr) then
    value := memi(loghead(48, addr), x86)
    return (nil, value , x86)
 else
    return (error, 0, x86)
 endif

wvm08-user-mode(addr, val, x86):
  // Write a byte "val" to virtual memory at address "addr"
  // Returns an error flag (nil if no error) and x86 state
  if canonical-address-p(addr) then
     x86 := !memi(loghead(48, addr), val, x86)
     return (nil, x86)
  else
     return (error, x86)
  endif
```

The top-level linear memory functions `rm08` and `wm08` call these functions when the `user-level-mode` field is `t`.

6.3 Normalizing Memory Accesses

Our model offers several memory-related functions that read and write different sizes of data. In order to avoid a large number of theorems about these various functions, we follow an approach similar to that for accessors and updaters to the x86 state, as described previously in Section 5.3. We define normal forms for accessing and updating linear memory during reasoning: functions `rb` and `wb` respectively.

The linear memory accessor function `rb` takes a list of linear addresses, permissions, and the x86 state as input, and returns three values — an error flag, a list of bytes read, and resulting x86 state. The linear memory updater function `wb` takes a map of linear addresses to bytes and x86 state as input and returns two values — an error flag and resulting x86 state. These functions allow reasoning about memory to be done in terms of simple operations on lists like membership, disjointness, etc. instead of arithmetic involving address ranges. Clearly, these functions are not suitable for execution because they deal with lists and can cause expensive "consing" (i.e., allocation of memory on the heap). Again, we use `mbe` to great benefit here. The bodies of `rm08`, `wm08`, and other similar functions actually contain an `mbe`, where the `:logic` part calls `rb` or `wb` as appropriate, and the `:exec` part calls the user-level or system-level mode functions, depending on the value of the field `user-level-mode`.

In this way, we retain the benefit of using normal forms to access and update the x86 state so that the number of Read-over-Write and other theorems is small, without compromising on execution efficiency. Note that the behavior of any x86 instruction that accesses and/or updates the x86 state can be described by a fixed set of theorems about `xr`, `xw`, `rb`, and `wb`.

Chapter 7

Instruction Semantic Functions and Undefined Behavior

Given the model of the x86 state and memory management mechanisms, we can specify the behavior of each x86 instruction in terms of reads from and writes to the x86 state. Each x86 instruction is specified by an instruction semantic function. We obtain the specification of x86 instructions by consulting the Intel and AMD manuals, and running tests on x86 machines to check our understanding of these manuals. Specification elicitation is one of the most challenging aspects of building an x86 ISA model. However, once an instruction has been added to our x86 model, we gain confidence in its accuracy by performing co-simulations. We describe our model validation process later in Chapter 8.

In general, an instruction semantic function defines the behavior of a set of opcodes. This function receives the decoded portions of the instruction from the step function, uses them to determine the location of operands (if any), and fetches them from the x86 state. Depending upon the opcode, this function then calls an *operation specification function*, which specifies the behavior of that opcode. For example, the behavior of all the following opcodes of the `ADD` instruction — `0x00` to 0x05, `0x80` to 0x83 — is specified by a single operation specification function

that takes two operands and the flags as input and returns the result and modified flags as output. A benefit of this approach is that it makes maintenance easier by reducing code bloat. Instructions which fetch operands and write outputs using the same addressing methods share the same top-level instruction semantic function. The actual computation is done by calling the appropriate operation specification function inside that function. For example, the following opcodes have a common top-level instruction semantic function but different operation specification functions: `ADD` (`0x00`, `0x01`), `OR` (`0x08`, `0x09`), `ADC` (`0x10`, `0x11`), `SBB` (`0x18`, `0x19`), `AND` (`0x20`, `0x21`), `SUB` (`0x28`, `0x29`), `XOR` (`0x30`, `0x31`), `CMP` (`0x38`, `0x39`), and `TEST` (`0x84`, `0x85`). The instruction semantic function for these opcodes is presented in Appendix B.

A non-trivial part of defining instruction semantic functions is accounting for operations that involve undefined behavior, randomness, or non-determinism. Many x86 instructions write undefined values to certain components of the machine state. For example, according to the Intel manuals, the arithmetic flags (carry, parity, auxiliary, zero, sign, and overflow flags) are all undefined after the execution of an unsigned divide instruction (`DIV`). Instructions like `RDRAND` and `RDSEED` compute random values. A source of non-deterministic behavior in our framework is the system call implementation in the user-level mode — system calls are non-deterministic from the point of view of an application programmer.

These three kinds of behaviors are different in nature and usage. Undefined behavior is one resulting from unspecified operations and an implementation is considered to be correct no matter what the result of this undefined behavior is. Operations resulting in random values have a defined behavior, namely the generation of

values that lack predictability. Non-deterministic computations can exhibit different behaviors for different runs, even if given the same inputs. However, all of them have the following characteristic in common: encountering such operations during reasoning should force an exhaustive case analysis that accounts for all possible behaviors, while they should evaluate to an appropriate concrete value during execution. Thus, specifying such behaviors in our x86 ISA model is complicated by the requirement of supporting both reasoning and execution.

In this chapter, we present our solution to modeling such behaviors when they are inherent in the instruction itself in Section 7.1. We describe how we model non-determinism arising due the system call service in the user-level mode of operation of our x86 ISA model in Section 7.2, where we also discuss why our solution of the previous section is not fitting for this scenario. We conclude our discussion of instruction semantic functions in Section 7.3, where we list some instructions that are specific to the system-level mode of operation.

7.1 Undefined and Random Behavior in x86 Instructions

Our x86 ISA model should be capable of both formal analysis using symbolic data and execution using concrete data, given instructions that exhibit undefined or random behavior. We first discuss how we support reasoning about undefined behaviors.

For reasoning, an undefined value should possess the following property:

Indeterminateness The result of an equality test of an undefined value with an-

other value, defined or undefined, should be unknown.

An example of why indeterminateness of undefined values is important is as follows. If a `DIV` instruction (which leaves the arithmetic flags undefined) is followed by a `JGE` instruction (which transfers control to a given address if the sign flag and overflow flag are equal, otherwise the control goes to the instruction following `JGE`), our formal analysis should not be able to determine whether the jump occurred or not; for reasoning to proceed, we would have to consider both the cases. This alerts us to unreliable and potentially dangerous behavior that may be exhibited by programs.

Undefined behavior can be modeled by using 4-valued logic, where the unknown value `X` possesses the indeterminateness property. However, the use of 4-valued logic would double the memory footprint of our model — we would require at least two bits to represent every bit of the machine state because undefined values can propagate elsewhere (e.g., after a `DIV` instruction, a program might push the flags register onto the memory). Instead, we provide a pool of indeterminate values that can be tapped by instruction semantic functions when necessary.

Undefined values are provided by the function `undef-value`, shown below. The function `access-undef` reads the value stored in the `undef` field of our x86 model. `Create-undef` is a *constrained function*; the only thing known about its output is that it is an unsigned integer. After admitting the function `undef-value`, we make the `undef` field's native updater function, `update-undef`, *untouchable* [17] to ensure that `undef-value` is the only function that can modify this field. Also, we never use `create-undef` in any function other than `undef-value`.

```
undef-value(x86):
 undef-seed := access-undef(x86)
 new-undefined-val := create-undef(undef-seed)
 new-x86 := update-undef(1 + undef-seed)
 return(new-undefined-val, new-x86)
```

Every call of the function `undef-value` produces a value that is equal to `create-undef` invoked with the current value of the `undef` field, and the `undef` field is incremented every time `undef-value` is called. Since `update-undef` and `create-undef` are never used outside the function `undef-value`, there are no collisions among any calls of `create-undef`. The consequences of the constrained nature of `create-undef` and its unique arguments are as follows:

1. The result of an equality test between any two calls of `create-undef` is unknown.

2. The result of an equality test between a call of `create-undef` and any defined value is unknown.

Thus, `undef-value` gives us indeterminate values every time it is called.

Apart from modeling undefined flags, `undef-value` is also used to specify instructions whose outputs involve randomness. The `RDRAND` instruction is one such example. This instruction is used to obtain a random number from the random number generator hardware, and then store it in a machine register. When the `RDRAND` instruction leaves the carry flag cleared, programs should disregard the value in the machine register. This situation indicates that the hardware was not ready

when `RDRAND` was invoked, and no data was transferred to the machine register. A set carry flag indicates that the value in the machine register is a valid random number. For specifying `RDRAND`, we invoke `undef-value` twice — once each for obtaining the random number and the value of the carry flag. This forces the user to reason about all possible values of the random number and the carry flag. Another example of an instruction where `undef-value` can be used for specification is `RDTSC` (read timestamp counter), which is used for performance monitoring. This instruction reads the value of the processor's time-stamp counter that monotonically increases every clock cycle and is reset whenever the processor is reset.

During execution, our model should behave exactly like an x86 processor — unlike during reasoning, we cannot account for all possible values. We re-define the logical ACL2 functions used for reasoning to run alternative code during execution. This is accomplished using an ACL2 feature called *trust tags* [133]. Trust tags allow arbitrary Common Lisp code to be defined outside ACL2 in raw Lisp. They indicate that this external code is *trusted* instead of *verified*. One way to support concrete executions is by arranging for this alternative code to perform *native execution*. For every instruction that requires an undefined or random value, we can define raw Lisp code to run the same instruction on the underlying x86 processor, obtain the values in the components that are supposed to be undefined, and return those values to the instruction semantic function. However, such context switches can slow down the execution speed of our x86 model significantly. Instead, for undefined flags, we use a suitable concrete value chosen based on tests performed on a real x86 machine. For example, we observed that the unsigned multiplication instruction (`MUL`) always

clears the zero flag, irrespective of its operands. According to the Intel manuals, the zero flag is undefined after the execution of the `MUL` instruction. Thus, we chose zero as the suitable value of zero flag for `MUL` during execution. For `RDRAND`, we simply invoke the underlying Lisp's `random` function twice, once each for the random number and the carry flag. Thus, even though an execution using our model that involves randomness or undefined behavior may not correspond to that on the actual machine, our model is still accurate in that it captures a behavior that is as possible as any other for that execution. We discuss how our setups for logic and execution correspond in the following section.

7.1.1 Relationship between Functions Used for Reasoning and Execution

It is important that our setup prohibits proofs of theorems that say that some undefined or random operation returns a specific value. Otherwise, we might be able to prove that the same operation returns some other value in some other ACL2 session. That could allow us to certify ACL2 books with contradictory theorems, and then include them both to prove `nil`, thereby causing a soundness violation. Similarly, the constrained function used for reasoning cannot be used for execution because it is non-executable. Therefore, the functions used for execution should not influence the reasoning process, and vice versa. We ensure this independence by making the following two arrangements:

1. When either of the two ACL2 flags, `in-prove-flg` or `in-verify-flg`, is true (that is, the model is being used for reasoning), specification functions call the functions used for reasoning.

2. When both `in-prove-flg` and `in-verify-flg` are false (that is, the model is being used for performing concrete executions), specification functions call the functions used for execution.

Note that our "official" x86 ISA specification that is used for reasoning consists of ordinary logic definitions that do not use trust tags. On the other hand, the setup for execution uses trust tags. This means that the official specification and the proofs done using that specification are not tainted by raw Lisp code from outside ACL2.

We illustrate our re-definition process by presenting the operation specification function of `RDRAND`. The function `HW_RND_GEN-logic` is the ACL2 definition that specifies the operation of `RDRAND` by calling `undef-value` twice.

```
;; ACL2 definition used for reasoning
(defun HW_RND_GEN-logic (size x86)
  (b* (((mv cf x86)
        (undef-value x86))
       (cf (logand 1 cf))
       ((mv rand x86)
        (undef-value x86))
       (rand (logand (1- (expt 2 (ash size 3))) rand)))
    (mv cf rand x86)))
```

The function `HW_RND_GEN` below simply invokes `HW_RND_GEN-logic`. Note that `HW_RND_GEN` is defined using ACL2's `defun-notinline` event [15], which is a directive to the underlying Lisp compiler not to inline calls of the function associated with `HW_RND_GEN`. This is important because re-defining inlined functions may result in unpredictable behavior. Another implication of using `defun-notinline` is that its

"real" name is HW_RND_GEN$notinline[1].

```
;; Top-level operation-specification function for RDRAND
(defun-notinline HW_RND_GEN (size x86)
  (HW_RND_GEN-logic size x86))
```

After introducing a trust tag, we re-define HW_RND_GEN$notinline so that it calls HW_RND_GEN-logic for formal analysis and raw Lisp code for concrete execution.

```
;; Raw Lisp definition used for execution
(defun HW_RND_GEN$notinline (size x86)
  (when
      (or
       (equal (f-get-global 'in-prove-flg ACL2::*the-live-state*)
              t)
       (equal (f-get-global 'in-verify-flg ACL2::*the-live-state*)
              t))
    ;; Code for formal analysis
    (return-from HW_RND_GEN$notinline
      (HW_RND_GEN-logic size x86)))
  ;; Code for execution
  (let ((cf
         (multiple-value-bind
             (cf random-state)
             (random 2)
           (declare (ignorable random-state))
           cf))
        (rand
         (multiple-value-bind
             (rand random-state)
             (random (expt 2 (ash size 3)))
           (declare (ignorable random-state))
           rand)))
    (mv cf rand x86)))
```

[1] ACL2 provides support for referencing foo$notinline using foo. A reader interested in details is referred to the documentation of the defun-notinline event [15].

Though our setups for reasoning and execution are independent (and necessarily so), they still specify the same behavior. What exactly is the (meta-)connection between them? The setup for execution characterizes one possible behavior described by the setup for reasoning. Consider two runs of our model with the same initial state, one in the reasoning setup and one in the execution setup. If the values returned by the logical functions (such as `HW_RND_GEN-logic`) match the values returned by raw Lisp functions (such as `HW_RND_GEN$notinline`) during execution, the final states obtained at the end of both the runs match.

7.2 System Call Service in the User-Level Mode

Application programs often make system calls to the underlying OS to request services like I/O and memory allocation. For example, the `printf` statement in C programs is ultimately implemented via a `write` system call. The most efficient and common way of invoking system calls on 64-bit x86 machines is through the x86 instruction `syscall`. The `syscall` instruction is used by application programs to call a system-level procedure at a higher privilege level; the value in the general-purpose register `rax` determines which system call is to be invoked. Its companion instruction, `sysret`, is used in the system-level procedure to return control to the application program — specifically, to the instruction after the caller `syscall` instruction.

In addition to the memory interface, the user-level and system-level modes differ in the specification of the `syscall` and `sysret` instructions. In the user-level mode, our model specifies the user-level x86 ISA and certain system call services. When verifying application programs in this mode, these services provided by the OS

are assumed to be correct. System-mode instructions (those that require the highest privilege level or CPL = 0), like `sysret`, `lldt`, `lgdt`, etc., are unavailable for use in the user-level mode.

In order to support system call service in the user-level mode, `syscall` instruction's semantic function has been extended to provide the semantics of system calls; we currently support `read`, `write`, `open`, `close`, `lseek`, `dup`, `link`, and `unlink` system calls. There are many often-used system calls that we do not yet support in our x86 ISA model — for instance, `mmap`, `exit`, `mprotect`, `madvise`, etc. More system calls can be added as and when required in the future. The specification functions of system calls are written in accordance with their `man` pages and other detailed descriptions [129, 137]. Note that the extended semantics of `syscall` are available in the user-level mode only; see Figure 7.1.

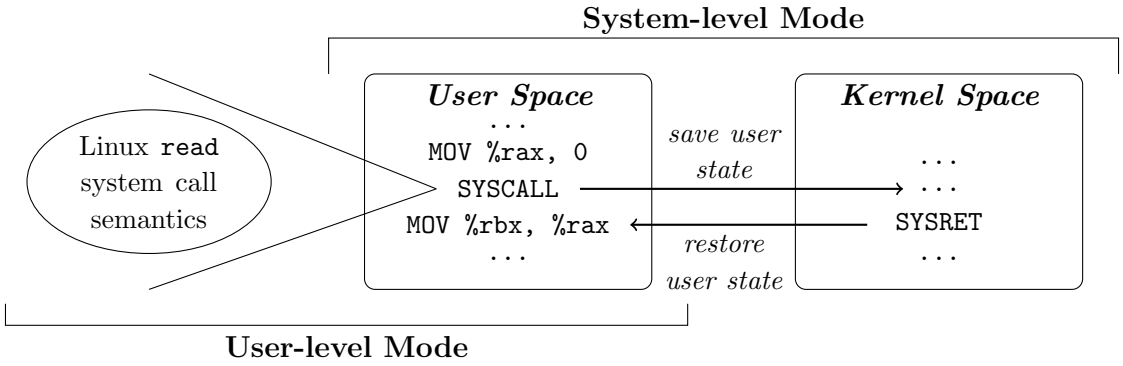

Figure 7.1: Treatment of Linux `read` System Call in Both Modes of Operation

Adding support for system calls in the user-level mode is non-trivial. Firstly, system call implementations differ among different OSes. A simple example is that the `read` system call is invoked on FreeBSD and Darwin machines if the `rax` register

has the value 3 and on Linux machines if it has the value 0[2]. Secondly, from the point of view of an application program, system calls are non-deterministic, i.e., different invocations of the same system call can give different results on the same machine. For example, the `open` system call may successfully open a file in one execution, but result in an error in another execution if that file has been deleted. Similar to our discussion in the previous section, the user-level mode should allow both formal analysis and concrete execution, given application programs that exhibit this non-determinism.

The first issue is solved by initializing a field in our x86 state, `os-info`, to identify the operating system under consideration; we support the Linux, Darwin, and FreeBSD systems. Depending on the value in this `os-info` field, the extended semantic function for `syscall` calls different functions corresponding to different system call implementations.

For execution, the second issue is solved by allowing our x86 model to directly interact with the underlying OS, using trust tags (as in the previous Section), to obtain the system call functionality. The `syscall` semantic function in ACL2 calls raw Lisp functions, which use a foreign function interface [23] to call C/Assembly functions. These C/Assembly functions make the required system call and return

[2]See the following files for more information on system call numbers:

- On Linux systems: `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`
- On FreeBSD systems: `/usr/src/sys/kern/syscalls.master`
- On Darwin systems: `/usr/include/sys/syscall.h`

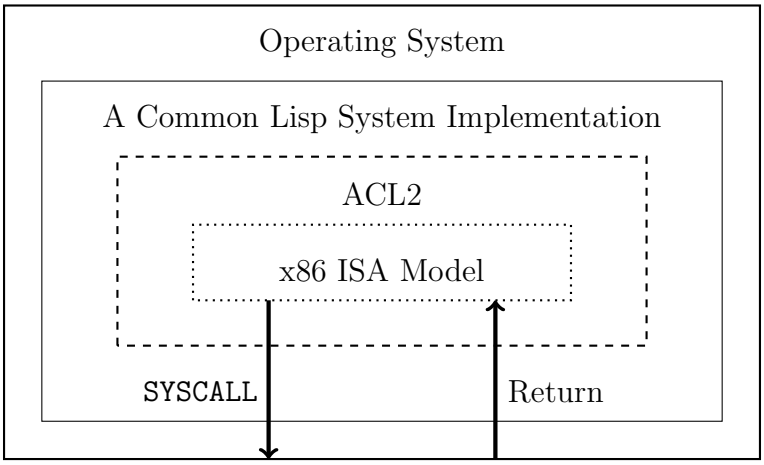

Figure 7.2: Execution Support for System Calls in the User-Level Mode

the results to the ACL2 caller function; see Figure 7.2 for an illustration. Of course, these raw Lisp functions should not be used for reasoning since they are *impure*: they depend on an external environment and hence, they may not always return the same results when evaluated with the same arguments.

For reasoning, the second issue is solved by consulting the environment field, `env`, in the x86 state. This field represents the part of the external world that affects or is affected by system calls. The field `env` models a subset of the *file system* and also consists of an *oracle* field. The oracle field specifies results of non-deterministic computations; it provides information that, though a part of the real environment, cannot be inferred from our model of the file system. An example is the file descriptor of a file to be opened — this descriptor is assigned by the OS depending on the number of files already opened for a particular process at the time the `open` system call is made. The oracle field maps linear addresses to a list of arbitrary values. Whenever the oracle is consulted, the list of values corresponding to the current

instruction pointer is located in the map. The first value in this list is taken for use and removed from the list. It is the responsibility of the user to initialize `env` (and hence, the oracle) appropriately. This is important because it allows the user to state explicitly what expectations are being made from the environment. Note that all the functions used for reasoning are *pure* — the return values only depend on the inputs and there are no observable side-effects.

We now use an example to describe how `env` allows reasoning about application programs that make system calls. Suppose that a program opens a file using an `open` system call, then writes to that file using `write`, and finally closes it using `close`. The `env` field needs to be initialized accordingly: the file system should contain the file's name and description, and the oracle field should associate the linear address of the first `syscall` instruction (i.e., `open`) with the value of a descriptor. Reasoning about this program will require reasoning about the following behavior (if no errors are encountered): the `open syscall` will pop off the descriptor from the oracle, associate it with a file name, and change the file's mode as requested; the `write syscall` will use the descriptor to locate the file and write the requested number of bytes from a specified memory buffer to the file; and, the `close syscall` will delete the association between the file descriptor and its name. The `env` field can be specified to contain symbolic as well as concrete information. For example, the file contents can be specified as a specific string if the user wished to reason about a program's interaction with a given file. If the user wished to characterize a program's behavior for all files or for files meeting some specific constraints (e.g., all files ending with the EOF character), the file contents can be specified using a

suitable symbolic string.

7.2.1 Relationship between Functions Used for Reasoning and Execution

The independence of functions used for reasoning and execution is maintained for the system call implementation using the same mechanism as discussed in Section 7.1 — the raw Lisp functions transfer control to the logical functions during reasoning.

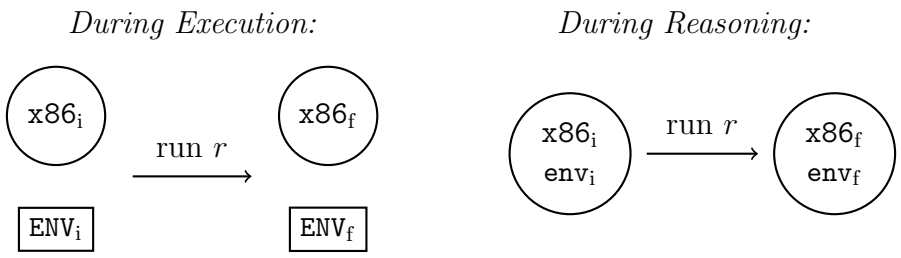

Figure 7.3: x86 States in Execution and Logic for System Calls: Let $\texttt{x86}_\text{i}$ be an x86 state. During execution, a run r takes an initial state $\texttt{x86}_\text{i}$ and returns $\texttt{x86}_\text{f}$, updating the external environment from $\texttt{ENV}_\text{i}$ to $\texttt{ENV}_\text{f}$. Then, during reasoning, the following holds: if $\texttt{env}_\text{i}$ corresponds to $\texttt{ENV}_\text{i}$, and $\texttt{x86}_\text{i}^\text{e}$ refers to $\texttt{x86}_\text{i}$ augmented with $\texttt{env}_\text{i}$, then the same run r from $\texttt{x86}_\text{i}^\text{e}$ during reasoning produces $\texttt{x86}_\text{f}^\text{e}$, which refers to $\texttt{x86}_\text{f}$ augmented with $\texttt{env}_\text{f}$, for some $\texttt{env}_\text{f}$ corresponding to $\texttt{ENV}_\text{f}$.

We now describe the connection between our setups for reasoning and execution of system calls in the user-level mode. Consider two runs of our model with the same initial x86 state, where one run is with the execution setup with the real external environment `ENV` and the other run is with the reasoning setup with the environment field `env` in the x86 state. The field `env` *corresponds* to the real environment `ENV` if the execution of system calls produces the same results while reasoning as those produced during execution. Figure 7.3 describes this correspondence more formally.

When this correspondence holds, each program run during execution produces a theorem under suitable hypotheses about the well-formedness of the `env` field. In this case, observations made about an application program during execution hold during reasoning, and vice versa.

Do we know whether this correspondence holds at all? This is established via co-simulations, i.e., by comparing concrete program executions using impure functions to corresponding evaluations in logic using pure functions and an appropriately initialized `env` field. The pure functions form the specification for the behavior of impure functions.

We demonstrate our co-simulation process for system calls using a short example. Consider the following assembly program that requests the `read` system call service from the underlying OS — Linux in this case — to read one byte from a file with descriptor 0 (standard input, usually). Arguments to the `read` system call are located in predetermined registers, according to x86-64 Application Binary Interface [139]. The register `rax` is initialized with the Linux read system call number, `rdi` with the file descriptor, `rsi` with the memory buffer's address where the read byte will be stored, and `rdx` with the number of bytes to be read.

```
mov $0x0,%rax         /* Syscall number   */
xor %rdi,%rdi         /* File descriptor  */
mov -0x20(%rbp),%rsi /* Buffer address   */
mov $0x1,%rdx         /* Number of bytes  */
syscall
```

If no error occurs, the raw Lisp function for the `read` system call reads a byte from the specified file and writes it to the memory buffer at `rsi`. During reasoning,

we initialize the environment field `env` so that the contents of the file with descriptor 0 contains the byte that was read in the corresponding run during execution. The rest of the x86 state is initialized to be exactly the same as the initial state during execution. We then evaluate the logical functions to simulate these five instructions. We validate our system call model by comparing the final state obtained here with that during execution.

7.2.2 `undef` vs. `env` Fields

Note that one can also use the oracle in the `env` field to specify undefined or random values, thereby making the `undef` field unnecessary. However, there is a fundamental difference in the way these two fields are used. The oracle field has to be initialized appropriately by the user, and this initialization provides a way of explicitly acknowledging reliance on an external environment. Such computations do not happen often. On the other hand, the behavior modeled by the `undef` field is a part of the ISA specification itself — there are no expectations from the external environment. Such undefined values are required often, for example, in the case of commonly-used instructions like `DIV` and `MUL`, which leave some flags undefined. Imagine having to initialize the oracle field whenever these instructions are executed.

Thus, though `env` and `undef` fields provide similar capabilities, they serve different purposes. We model non-deterministic behavior stemming from reliance on an external environment using the `env` field and the ISA-specified undefined or random behavior using the `undef` field. Such design decisions are in the interest of practicality — they spare the user from performing extra work for both program

specification and verification.

7.3 Instructions Specific to the System-level Mode

System-mode instructions are available for use in the system-level mode of operation. These include a specification of 64-bit segmentation-related privileged instructions like `lgdt` (load global descriptor table register) and `lldt` (load local descriptor table register); these instructions load the selector field in the `gdtr` or `ldtr` system register with their source operand after performing descriptor-table limits and checking the validity of the segment selector and descriptor. The specification of `sysret`, unavailable in the user-level mode, is also included in the system-level mode, and `syscall` is specified exactly as described by the Intel manuals.

Note that if making the assumption of correctness of OS services is undesirable, then instead of the user-level mode, application programs can be verified in the system-level mode. System programs, like kernel routines, must necessarily be verified in the system-level mode because the user-level mode does not expose the x86 ISA's system state.

Chapter 8

x86 ISA Interpreter and Model Validation

In order to support both concrete and symbolic simulation of x86 machine-code programs, our x86 ISA model must be capable of fetching x86 instructions from the memory in the x86 state, decoding them, and then executing them by calling the instruction semantic functions. That is, we need an x86 ISA interpreter. Specifying the fetch-decode-execute cycle for x86 machines is non-trivial, given that x86 instructions are of variable length and the syntax rules governing their encoding are lengthy and complicated. Intel manuals devote around 150 pages (in Appendices A and B of Volume 2 of Intel Manuals [40]) just to describe opcodes, instruction formats, and encodings.

Therefore, in addition to our instruction semantic functions, we need to ensure that our specification of the IA-32e fetch-decode-execute cycle is free from modeling errors. We gain confidence in the accuracy of our x86 ISA model by supplementing code reviews with *co-simulations*. Co-simulation is the process of executing a machine-code program on the processor as well as on the model, and then comparing their resultant states after every instruction. If their states match, then the model is known to be accurate for at least those instructions and data encountered during that run. Clearly, the key to gaining more confidence in the model's accu-

racy is to perform a large number of co-simulations with different kinds of data and instructions.

In this chapter, we present our step function that executes one x86 instruction in Section 8.1 and the run function that can execute a specified number of x86 instructions in Section 8.2. In Section 8.3, we describe our process for validating our x86 ISA model via co-simulations, including the tools we developed to aid in this task.

8.1 Step Function

The step function, called `x86-fetch-decode-execute`, takes an x86 state as input and returns an x86 state that captures the effects of executing one x86 instruction. The instruction pointer register `rip` contains the linear address of the next instruction to be executed. The x86 ISA model's step function fetches an instruction located at this address in `rip` from the model's memory, decodes it, and dispatches control to the appropriate instruction semantic function. See Appendix C for the definition of `x86-fetch-decode-execute`.

The maximum length of a legal x86 instruction is 15 bytes, but most instructions are much smaller than this limit. In the interest of execution efficiency, the step function lazily fetches one byte of an instruction at a time. It fetches just enough bytes so that the instruction semantic function can be called; i.e., it fetches the prefixes, opcode, and the `ModR/M` and `SIB` bytes (if any). The instruction semantic functions fetch the address displacement and the operand(s), if any. The step function can result in an x86 state that has non-empty `fault` or `ms` fields if

0 or more		1, 2, or 3 bytes	0 or 1 byte	0 or 1 byte	0, 1, 2, or 4 bytes	0, 1, 2, or 4 bytes
Legacy Prefixes	Rex Prefix	Opcode	ModR/M	SIB	Address Displace-ment	Immediate Operand

Figure 8.1: x86 Instruction Format in IA-32e Mode

it encounters unimplemented behavior (e.g., exceptions or an unsupported opcode). An x86 state with non-empty values of `fault` or `ms` fields is in a "halt" state, and we offer no guarantees that such a state reflects the real machine. See Section 5.1 for a description of these two fields in the x86 state.

The format of an x86 instruction is depicted in Figure 8.1. An x86 instruction consists of optional *prefixes* that can be used to modify the default attributes of the instruction such as the size of the operand(s), *opcode(s)* that identify the instruction, and an *addressing-form specifier* that locates the operand(s) of the instruction (if any). The addressing-form specifier consists of the ModR/M byte and one or more of the following (if necessary): SIB byte, an address displacement field, and an immediate data field. An x86 instruction is variable-length — each byte in an x86 instruction gives information about the nature or number of the next byte(s). Intel's opcode maps give the addressing information for each machine opcode. For example, the 0x00 opcode for the `ADD` instruction is specified to have the encoding `Eb, Gb`, and the Intel manuals assign the following code for these letters [39]:

- `E`: "A `ModR/M` byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory ad-

dress, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement."

- `G`: "The `reg` field of the `ModR/M` byte selects a general register."

- `b`: "Byte, regardless of operand-size attribute."

Our x86 model contains tables that encode this textual information present in the Intel manuals in a machine- as well as a human-readable format. The step function uses these tables to aid in instruction decoding. This reduces interpretation errors and ensures that no simplification in x86 semantics is made inadvertently.

8.2 Run Function

The run function, `x86-run`, specifies our x86 ISA interpreter. This function takes an x86 state and the number of instructions to be executed as input, and returns an appropriately modified state as output. Execution is halted when the upper limit on the number of instructions to be executed is reached or if an unrecoverable error is encountered, as indicated by non-empty `ms` or `fault` fields in the x86 state.

The definition of the run function is straightforward, as shown below. It simply calls the step function recursively.

```
(defun x86-run (n x86)
  ;; Halt if there is a problem indicated by the ms or
  ;; fault fields, or if there are no more instructions
  ;; left to execute.
  (cond ((fault x86) x86)
        ((ms x86) x86)
```

```
((zp n) x86) ;; Case n is 0
(t (b* ((x86 (x86-fetch-decode-execute x86))
        (n (1- n)))
     (x86-run n x86)))))
```

8.3 x86 ISA Model Validation

We obtain the specification of x86 ISA by referring to the Intel manuals, cross-referencing AMD manuals, and running tests on x86 machines. Clearly, this is an error-prone process, with the possibility of making subjective judgments about the processor's behavior. How then do we know that our specification accurately captures the behavior of x86 machines? This is of paramount importance — after all, any analysis done using our framework is valid only if our x86 model is an accurate representation of the x86 ISA. We validate our model via code reviews and by performing co-simulations against an x86 machine.

Our approach to performing co-simulations is depicted in Figure 8.2. Given a machine-code program in an executable file format, we determine if our model supports enough instructions and features to execute it; if not, we implement them. We parse the executable file and retrieve information about where the program should be placed in the linear memory. After loading the program into the model's memory and initializing other parts of the x86 state appropriately, we execute it both on our model and on the real machine. Using program instrumentation tools, we log the states of the model and the machine at a desired level of granularity, i.e., on a per-instruction basis or at certain breakpoints. If the model and machine agree, then we know that at least for those instructions and data, our model accurately represents

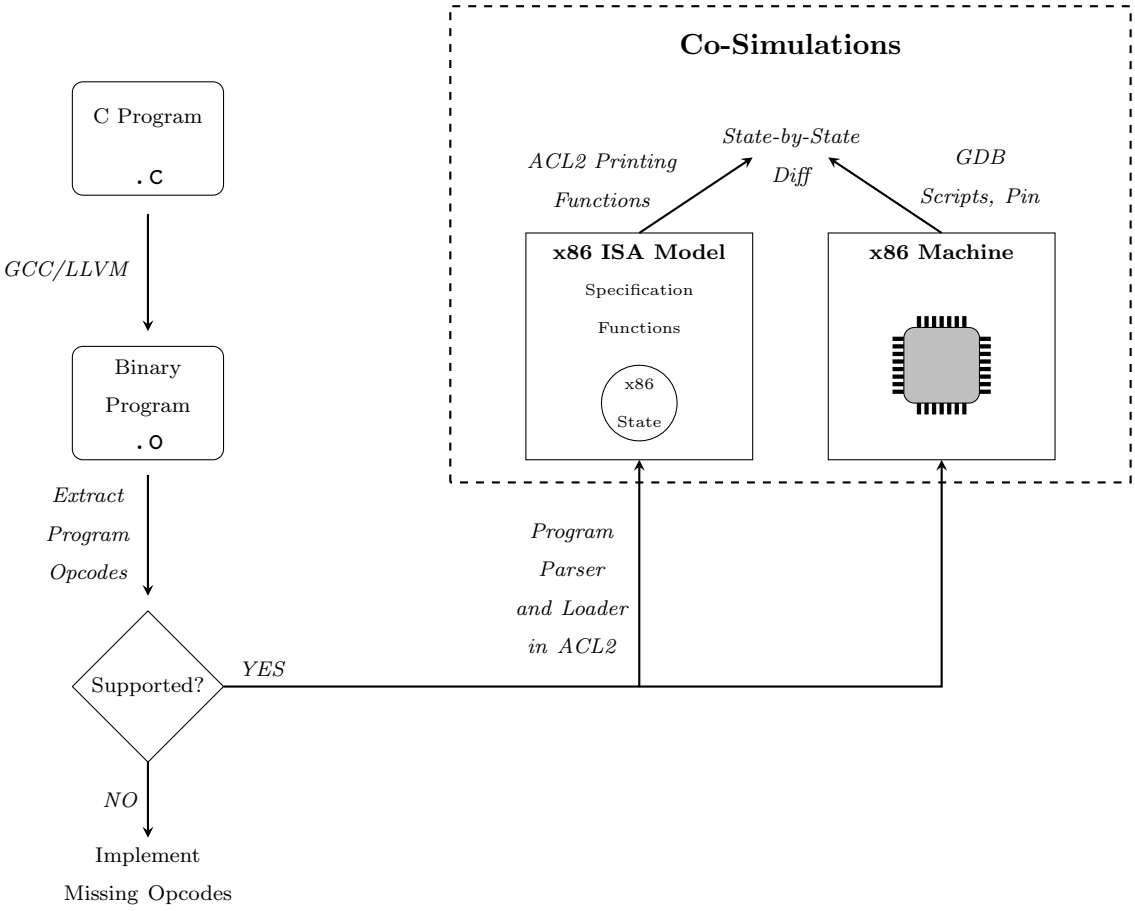

Figure 8.2: x86 ISA Model Validation via Co-simulations

the x86 ISA. Any disagreement indicates the presence of a bug in our model; we then localize the source of this difference to fix errors in our specification. Note that our co-simulation framework can also be used to detect hardware bugs in x86 processors or inaccuracies in the ISA specifications released by hardware vendors.

Recall that upon encountering ISA features that are unimplemented in our x86 model, the `ms` or `fault` fields are set in our x86 state, indicating that a model-related error was encountered. If, for instance, the program used for co-simulations causes an exception, the co-simulation halts and our model reports the reason — namely, exceptions are currently unsupported by our x86 ISA model.

We developed three ACL2-based tools that are central to our co-simulation process:

1. **x86 Machine Program Parser and Loader:** This tool supports Mach-O [124] file format for Darwin Systems and ELF [138] for Linux and FreeBSD systems. The parser reads the executable file, determines if it is well-formed, and uses the header structures and load commands present in it to decode various sections, like text and data. The loader then loads these sections at appropriate linear addresses in the memory of our x86 state.

2. **x86 State Initialization Functions:** It is essential for the model's initial state to match the machine's state when the machine is poised to execute the program. This is to ensure that spurious differences due to improper model state initialization do not slow down our validation process; if there are no bugs in our model and if the initial states of the model and the machine match,

then only will the model and machine be in lock-step with each other. Note that the parser and loader only load the program into the memory, and we provide convenient ACL2 functions to initialize other components of the x86 state as well. In the system-level mode of operation, our x86 state initialization functions also provide a default configuration of paging data structures — 1G pages with an identity map of linear to physical addresses.

3. **Program Instrumentation:** We use GNU Debugger [33] and Intel's Pin [74] to instrument programs and log the state of an x86 machine. We have developed analogous tools for our x86 model; these tools provide a large range of capabilities. For example, we can trace every read from or write to the x86 state, including the entire memory, and this tracing can be conditional, where the condition can be stated using an arbitrary ACL2 function. We can set breakpoints in our program that temporarily halt its execution; we can even modify the program at any point during the execution. Thus, our model is a programmable simulator, which allows a program to be executed in a safe environment to study its behavior.

In addition to maintainability, a benefit of developing these tools in ACL2 is that, if necessary, we can reason about them to obtain assurances about their correctness.

Execution efficiency is essential for performing co-simulations. The higher the execution speed of our x86 model, the more co-simulations we can perform, and thus, the more confidence we will have in our model's accuracy. The execution speed of our model is around 3.3 million instructions/second in the user-level mode

and around 320,000 instructions/second in the system-level mode with a set-up of 1G page-table configuration[1]. We have used several programs as test cases — our largest application program is a contemporary SAT solver[2] and our largest system program is the *zero-copy* program described later in Chapter 12.

Some families of Intel CPUs have been known to have design defects or bugs, which cause the processor's behavior to deviate from their stated specifications. The known design defects are published by Intel in "specification update" documents [41]. Also, different families may have different defects, which means that some instructions may behave differently on different Intel processors. Ideally, an x86 ISA specification should model all processor families. In this research, our target for co-simulations is an Intel Xeon E31280 CPU.

Our x86 ISA model can also be validated against instruction-set simulators like QEMU [90]. This has the benefit of examining both our specification as well as the implementation of such mainstream simulators. In fact, a member of the ACL2 community has been performing such co-simulations, as a result of which some bugs in our instruction decoding functions were found and fixed [52].

To ensure the efficacy of our co-simulations, we must run them with suitable tests that cover as many behaviors of instructions as possible — e.g., tests that exercise every branch of our instruction semantic functions. At this time, we have chosen test programs more or less randomly, though we have run millions of tests

[1]This speed was measured on a Intel Xeon E31280 CPU @ 3.50GHz with 32GB RAM.

[2]This solver, developed by Marijn Heule, provides performance similar to state-of-the-art solvers.

with judiciously chosen inputs that represent certain classes of possible input values. For example, when validating our specifications of floating-point instructions, we ran tests with signed zeros, denormalized finite numbers, normalized finite numbers, signed infinities, NaNs, and indefinite numbers. This process can certainly be more streamlined in the future. We can envision using a Lisp code coverage tool [22] that can annotate untested lines, thereby helping us construct a better test suite. Another way would be to generate automatic test cases from the instruction specification functions in our x86 ISA model, similar to the recent work [69] by Campbell and Stark.

Part III

Program Analysis Libraries

Chapter 9

General Strategy for x86 Machine Code Analysis

Formal verification of machine code can be an arduous process — it involves reasoning about complicated mechanisms like memory management and the fetch-decode-execute cycle, as well as accesses and updates to the machine state, which contains a multitude of registers, flags, and a large memory. We have developed a program verification framework, consisting of general-purpose lemma libraries, that reduces the time and manual effort required to prove properties of machine-code programs using our x86 ISA model. These libraries are intended to be included in every program verification project done using our framework. Broadly speaking, they consist of lemmas that axiomatize the behavior of x86 instructions. These lemmas are generally stored as ACL2 *rewrite rules*[1], which allow rewriting an expression to another generally simpler expression. These rules can *fire* (i.e., be applied) automatically, thereby enabling largely automatic symbolic simulation of x86 programs. There may be instances when these rules fail to fire — we discuss how we can debug failed applications of rules later in this chapter.

How do we begin building these general-purpose lemma libraries? Performing program analysis and developing lemma libraries are inter-dependent tasks. Our

[1]There are some exceptions — for instance, some lemmas are stored as *congruence rules*. More on that in Chapter 10.

usual approach is to discover useful lemmas about our specification functions during attempts to reason about programs, which may then suggest similar lemmas involving other functions. Then we add these lemmas to our libraries, after generalizing them as needed. Clearly, verifying different kinds of programs will help in the discovery of different kinds of general lemmas that can be re-used for the analysis of other programs.

In this chapter, we present some typical lemmas included in our code proof libraries and then describe our basic strategy for program analysis. We present ACL2 rules that control symbolic simulation of programs in Section 9.1. In Section 9.2, we use an example program to demonstrate how our lemma libraries facilitate two key undertakings: capturing the effects of a program and discovering preconditions under which a program behaves as expected. For demonstration purposes, this program is extremely small and simple. Also, for simplicity, we choose the user-level mode of operation for this example. We postpone the discussion of the special considerations needed when analyzing supervisor-mode programs to Chapter 10, where we use this same simple program to contrast the differences between these two modes of operation. In Section 9.3, we discuss some important issues to keep in mind in order to obtain a "reasonable" set of preconditions. Finally, Section 9.4 supplements this chapter with a brief discussion of the challenges of porting over a program proof performed at the level of source code to that at the level of machine code.

It is worth mentioning that this chapter can be viewed as a description of how users typically interact with theorem provers, specifically ACL2, in order to prove properties about their specification functions. Readers new to interactive theorem

proving may find this description illuminating, while readers with some experience may find this chapter relatively familiar.

9.1 Controlling Symbolic Simulation

Traditionally, code verification efforts in ACL2 that employ machine models follow a simple strategy [102] to automate and control symbolic simulation of programs. This strategy involves proving three main rules about the interpreter under consideration. We describe these rules in the context of the x86 ISA model. Note that these rules are available in both modes of operation of the x86 model.

1. **Step Function Opener Rule:** This rule specifies the conditions under which a call of the function `x86-fetch-decode-execute` should be *expanded* by the theorem prover. That is, it determines when a call of this function should be replaced by its body during reasoning. One of these conditions is that to begin with, the `ms` and `fault` fields should signal a non-erroneous x86 state. Another condition is that each instruction byte should be located at a canonical linear address. We disable `x86-fetch-decode-execute` after proving this rule so that ACL2 cannot use its definition for expansion during proof attempts. Thus, this opener rule is the only way calls of the step function are expanded during reasoning.

   The step function opener rule imposes an order in which the calls of the step function are expanded — it forces ACL2 to first expand that call about which pertinent information to resolve the conditions is known. Typically, this means expanding the innermost call, corresponding to the current

instruction, in a nest of calls of the step function. This is important because letting all calls of the step function expand indiscriminately will overwhelm the theorem prover with cases. Recall that the step function calls all the instruction semantic functions, and indiscriminate expansion will result in a case analysis of *all* possible instructions at *every* step.

If the conditions of this rule are not resolved, it means that the instruction or the state is ill-formed in some way, or sufficient information is not known (e.g., some rules are disabled), or we have a missing precondition. In this case, the step function will not be expanded and no progress will be made vis-à-vis symbolic simulation.

2. **Run Function Sequential Composition Rule:** This rule allows ACL2 to rewrite expressions matching

   ```
   x86-run(clk+(n1, n2), x86)
   ```

   to

   ```
   x86-run(n2, x86-run(n1, x86))
   ```

   where `clk+` is simply the addition function. This is an important rule because it facilitates compositional reasoning; it reduces the problem of reasoning about `(n1 + n2)` instructions to two smaller problems — first reasoning about `n1` instructions, and then about `n2` instructions. More formally speaking, if `(n1 + n2)` instructions transform a given x86 state from $\texttt{x86}_\text{i}$ to $\texttt{x86}_\text{f}$, then the effects of these instructions can also be described by first symbolically simulating `n1` instructions given $\texttt{x86}_\text{i}$ and obtaining an intermediate state $\texttt{x86}'$, and then simulating `n2` instructions given $\texttt{x86}'$ and obtaining state $\texttt{x86}_\text{f}$.

3. **Run Function Opener Rule:** This rule controls the expansion of the run function. It corresponds directly to the definition of `x86-run` presented previously in Section 8.2. A call of the run function `x86-run(n, x86)` is rewritten to `x86` if an error is signaled by the `ms` or `fault` fields in the x86 state or if all the instructions have been executed (`n == 0`). Otherwise, the call is rewritten to `x86-run(n - 1, x86-fetch-decode-execute(x86))`. We disable the run function after proving this opener theorem.

These three rules above control the "unwinding" of the x86 interpreter, but we still need to provide rules to enable reasoning about the effects of each instruction. We had described these rules previously in Chapter 3 — Read-over-Write, Write-over-Write, Writing-the-Read, and State Well-Formedness Theorems. These rules axiomatize the behavior of x86 instructions by describing the interaction between read and write operations involving the x86 state. Examples of these rules involving the state accessors `xr` and updater `xw` are presented in Appendix D.

Note that all these rules that control symbolic simulation are true for *any* program that runs on our x86 ISA model. Moreover, these rules are in terms of our x86 interpreter, and as such, describe and axiomatize the x86 ISA as well.

9.2 Example: Describing A Program's Effects

We now present a small, contrived example to describe our modus operandi of using these rules to capture the effects of instructions. We use the user-level mode of operation of the x86 ISA model for this illustration. Consider the following

machine program snippet that we represent as an ACL2 constant called `*program*`; this snippet contains two one-byte instructions.

```
(defconst *program*
  '(#xF8 ;; CLC instruction --- clears the Carry Flag
    #xF9 ;; STC instruction --- sets the Carry Flag
    ))
```

In order to symbolically simulate this program, we specify the following rather obvious preconditions in ACL2:

```
(defun-nx preconditions-in-user-level-mode (x86)
  (and
   ;; The x86 state is well-formed.
   (x86p x86)
   ;; The model is operating in the user-level mode.
   (user-level-mode x86)
   ;; The program is located at linear addresses ranging from
   ;; (rip x86) to (+ -1 (len *program*) (rip x86)).
   (program-at (create-canonical-address-list
                (len *program*) (rip x86))
               *program* x86)
   ;; The addresses where the program is located are canonical.
   (canonical-address-p (rip x86))
   (canonical-address-p (+ -1 (len *program*) (rip x86)))
   ;; The initial state is error-free.
   (equal (ms x86) nil)
   (equal (fault x86) nil)))
```

Note that all the addresses specified above are symbolic, and that `(len *program*)` is equal to 2. Also, recall that canonical addresses are legal linear addresses — using non-canonical addresses causes an exception (General Protection Exception, `#GP(0)`) on x86 machines.

Suppose we wanted to see the effects of the first instruction only. We submit the following obvious non-theorem to ACL2.

```
(defthm program-effects-in-user-level-mode-1
  (implies (preconditions-in-user-level-mode x86)
           (equal (x86-run 1 x86)
                  ???)))
```

ACL2 will be unsuccessful in proving this theorem because the value of the expression `(x86-run 1 x86)` is certainly not equal to `???`. However, ACL2 will fail in a useful way. Given our rules, it will simplify `(x86-run 1 x86)` in the following manner, thereby capturing the effects of `CLC` in terms of updates done to the x86 state:

```
(x86-run 1 x86)
;; Using the run opener theorem
==
(x86-run 0 (x86-fetch-decode-execute x86))
==
;; Using the step opener theorem and expanding the CLC
;; instruction semantic function
==
(x86-run 0 (!rip (+ 1 (rip x86)) (!flgi *cf* 0 x86)))
==
;; Using the run opener theorem
(!rip (+ 1 (rip x86)) (!flgi *cf* 0 x86))
```

Submitting the following modified version of `program-effects-in-user-level-mode-1` to ACL2 will succeed.

```
(defthm program-effects-in-user-level-mode-1
  (implies (preconditions-in-user-level-mode x86)
           (equal (x86-run 1 x86)
                  (!rip (+ 1 (rip x86)) (!flgi *cf* 0 x86)))))
```

Similarly, we hope to obtain the effects of both the instructions by submitting the following to ACL2.

```
(defthm program-effects-in-user-level-mode-2
  (implies (preconditions-in-user-level-mode x86)
           (equal (x86-run 2 x86)
                  ???)))
```

Unfortunately, instead of giving us a "clean" expression (i.e., one that is in terms of updates made to the initial state), ACL2 gives us the following expression.

```
(X86-FETCH-DECODE-EXECUTE
 (XW :RIP 0 (+ 1 (XR :RIP 0 X86)) (!FLGI *CF* 0 X86)))
```

This expression indicates that the step opener rule did not fire successfully for the second instruction. In order to understand why the opener rule failed, we use one of ACL2's proof debugging features — `break-rewrite` [14]. This feature allows the user to monitor rules and access contextual information as they are being tried for application by the rewriter. Upon monitoring our step opener rule, ACL2 tells us that the attempt to apply it failed because a hypothesis of that rule rewrote to the following expression instead of to `t` — that is, this hypothesis could not be relieved. Note that the syntactic form of the expression below is one used internally by ACL2.

```
(CANONICAL-ADDRESS-P (BINARY-+ '2 (XR ':RIP '0 X86)))
```

This information tells us that our preconditions are incomplete because they only state that addresses `(rip x86)` to `(+ -1 (len *program*) (rip x86))` are canonical. We also need `(+ 2 (rip x86))` to be canonical because the second instruction advances the instruction pointer to point to this address. Thus, we modify

`preconditions-in-user-level-mode` as follows; note that we only changed the fifth precondition.

```
(defun-nx preconditions-in-user-level-mode (x86)
  (and
   ;; The x86 state is well-formed.
   (x86p x86)
   ;; The model is operating in the user-level mode.
   (user-level-mode x86)
   ;; The program is located at linear addresses ranging from
   ;; (rip x86) to (+ -1 (len *program*) (rip x86)).
   (program-at (create-canonical-address-list
                (len *program*) (rip x86))
               *program* x86)
   ;; The addresses used in the program are canonical.
   (canonical-address-p (rip x86))
   (canonical-address-p (+ (len *program*) (rip x86)))
   ;; The initial state is error-free.
   (equal (ms x86) nil)
   (equal (fault x86) nil)))
```

Now, ACL2 successfully gives us the symbolic expression corresponding to executing this program.

```
(defthm program-effects-in-user-level-mode-2
  (implies (preconditions-in-user-level-mode x86)
           (equal (x86-run 2 x86)
                  (!rip (+ 2 (rip x86)) (!flgi *cf* 1 x86)))))
```

The right-hand side of the conclusion of the above theorem describes the resulting instruction pointer in terms of the initial instruction pointer, which was a symbolic value. Also, note that ACL2 did not simplify `(x86-run 2 x86)` to the following symbolic expression, which though still correct, is confusing at first glance.

```
(!rip (+ 2 (rip x86))
      (!flgi *cf* 1
             (!rip (+ 1 (rip x86)) (!flgi *cf* 0 x86))))
```

This is because of the Write-over-Write rules that eliminate shadowed writes, thereby making output from ACL2 more user-friendly.

This is, of course, an exceedingly simple example, but it illustrates how we discover preconditions needed to symbolically simulate a program. The output from failed proof attempts suggests which rules are not being applied successfully, and monitoring those rules gives us insights into which preconditions are missing. The documentation of our analysis framework lists some common rules to monitor in case of failed proof attempts [49].

9.3 Considerations for Specifying Preconditions

An important issue when discovering and specifying preconditions is to ensure that they are not too restrictive, or worse, contradictory[2]. Analysis of a program given restrictive preconditions takes only a small set of program runs into account, thereby decreasing the applicability of formal analysis. Contradictory preconditions render the formal analysis completely worthless — given a false antecedent, the truth value of an implication is always `T`, and thus, *any* program property can be derived given such preconditions.

There is a simple sufficient condition for our preconditions to be contradictory, namely, that ACL2 can prove `nil` from these preconditions. However, in practice,

[2]Of course, this is an important consideration for all theorems, in general.

this strategy may not work because ACL2 might not know enough about these predicates to infer that they are contradictory. This might happen if the predicates are complicated, or if the user directs ACL2 to turn off some reasoning strategies and/or to *disable* the functions or theorems describing these predicates. A contrived but illustrative example is as follows.

```
;; Succeeds --- note that (atom x) == (not (consp x))
(thm (implies (and (consp x) (atom x)) nil))

(defun predicate-1 (x) (consp x))
(defun predicate-2 (x) (atom x))

(in-theory (disable
            ;; Disabling the functions so that ACL2 cannot
            ;; "see" their definition during reasoning
            predicate-1
            predicate-2
            ;; Disabling a decision procedure
            (:executable-counterpart tau-system)))

;; Fails --- ACL2 does not know enough about the preconditions
(thm (implies (and (predicate-1 x) (predicate-2 x)) nil))
```

Of course, one can manually inspect the preconditions to ensure that they are reasonable. However, if establishing a program's property requires assuming a large number of preconditions (the zero-copy program presented in Chapter 12 is such an example), manual inspection will be laborious and error-prone.

To solve this problem, we use our x86 ISA model's capability to perform concrete tests. The preconditions for a program's correctness describe a symbolic x86 state that captures all relevant initial states where the program is poised to begin

execution. We concretize this symbolic state by initializing our model's x86 state with suitable concrete values — that is, we create a *witness* for this symbolic state. Then, we evaluate the functions describing the preconditions with this witness as input. If these functions return `nil`, then the witness does not satisfy the preconditions; this indicates that either the preconditions are contradictory, or that the witness is not representative of the preconditions. Otherwise, if the functions return `t`, we know that the preconditions are not contradictory, which gives us confidence that they are reasonable. One can perform these tests for a variety of witnesses (i.e., states with different locations of the program, stack, and data in the memory, different values in the registers and flags, and so on).

A user might also be interested in determining a minimal set of preconditions for which a program property holds. Though our reasoning framework does not directly provide a capability to do that, it should be noted that our process of adding preconditions incrementally helps in limiting their number. Another option, which is frequently employed by us during code proofs, is to use an ACL2 tool called `remove-hyps` [16] to remove unnecessary preconditions in our theorems. Given a theorem, this tool attempts to obtain a stronger theorem by removing one hypothesis at a time and checking if the resulting conjecture is still provable.

9.4 Porting Proofs to the Level of Machine Code

It is worth mentioning here that if a program was verified at the level of source code, porting its proof of correctness over to the level of machine code may not be straightforward. There may be a considerable difference between the state-

ments of properties expressed at the level of source code and that at the level of machine code. The former may be more "compact" than the latter, simply because programmer-friendly abstractions can be used to make a statement about the behavior of high-level programs — these abstractions cease to exist for machine-code programs. For instance, if a program uses compound data structures, then a property can be expressed in terms of fields of those data structures during high-level code analysis. For machine-code analysis, the same property would have to be expressed in terms of memory locations. Also, proofs of high-level programs may not apply directly to their corresponding machine-code programs. One reason is that the verification strategies for source and machine code analysis are considerably different. For example, a field `A` in a compound data structure is *known* to be distinct from another field `B` in the same data structure — we trust that this separation is provided correctly by the programming language implementation. For machine-code analysis, we would have to explicitly establish the non-interference of the memory locations corresponding to fields `A` and `B`. Another reason is that properties proved at specific statements of high-level programs may not hold at the machine instructions corresponding to those statements because of re-ordering optimizations by the compiler.

Chapter 10

Strategies for Reasoning about Paging

Supervisor program verification is more involved than application program verification, partly because a larger x86 state is exposed to these programs. In this chapter, we describe our approach to making supervisor program verification tractable. We focus on how the paging mechanism affects code proofs in the system-level mode of operation of the x86 ISA model.

We urge the reader to refer back to Chapter 6 for a reminder of how IA-32e paging works. Figures 6.3, 6.4, and 6.5 are especially useful because they show the possible configurations of paging data structures in the IA-32e mode. A reason why working with these data structures is complicated is that they are hierarchical, with two to four levels of indirection, depending on the page configuration. Another reason is that the format of each entry in the paging data structures is different based on its role in address translation. Let us consider a `PDPTE` entry (entry of a Page Directory Pointer Table) as an example. If this `PDPTE` entry is invalid, it is ignored. If it is valid and points to a page directory, then its bits 12 to 51 form the base address of the page directory. If, instead, it maps a 1GB page, then its bits 30 to 51 form the base address of the page frame. Yet another reason is that two special fields in the 64-bit paging entries — bit 5, the *accessed flag* (`A` flag) and bit 7, the *dirty flag*

(D flag) — are updated by the processor during paging structure traversals, thereby causing side-effects. The implications of the updates to these two flags are the main topic of this chapter.

The A flag is present in every entry used in any address translation. The D flag is present only in those entries which map a page (e.g., a PDPTE has a D flag field only if it maps a 1GB page). Whenever an entry is referenced during a traversal of the paging structures for address translation, the processor sets its A flag. When the address translation is done on behalf of a memory write operation (i.e., a write is to be done at the linear address being translated), then the processor sets the D flag in the final paging entry that maps the page. Effectively, A and D flag updates *mark* the translation-governing entries of a linear address being translated.

Thus, every linear memory read operation — including instruction fetches — can change the memory (because of updates made to A flags in the relevant paging entries), and hence the x86 state. Similarly, every linear memory write operation can cause additional writes to the memory (because of updates made to A and D flags in the relevant paging entries), and hence the x86 state. This leads to *cluttered* symbolic expressions during reasoning about supervisor code. We illustrate what we mean by this using an example in Section 10.1. In Section 10.2, we describe how these side-effect updates increase reasoning overhead, which prompted us to devise two sub-modes of operation of the system-level mode in order to enable effective reasoning. We also discuss how we used ACL2's congruence-based rewriting [11, 67, 130] to effectively ignore these side-effect updates during reasoning when they do not influence the behavior of the program under consideration.

10.1 Example: Describing A Program's Effects

The example program under consideration is the same as that previously discussed in Chapter 9, but now, we obtain its effects in the system-level mode instead of the user-level mode of operation. Note that for this simple program, the only interactions with the linear memory are read operations to fetch its two instructions.

```
(defconst *program*
  '(#xF8 ;; CLC instruction --- clears the Carry Flag
    #xF9 ;; STC instruction --- sets the Carry Flag
    ))
```

The preconditions for symbolically simulating `*program*` in this mode are shown below. The function `las-to-pas` specifies the address translation mechanism — its inputs are a list of linear addresses, their origin (i.e., whether the translation request originated from a memory read `:r`, or write `:w`, or instruction fetch `:x` operation), and an x86 state, and its outputs are an error flag, physical addresses corresponding to the input linear addresses, and an x86 state whose physical memory has been modified to account for `A` and `D` flag updates of the relevant paging entries. Also, ignore the third predicate, `(page-structure-marking-mode x86)`, present in these preconditions — we will postpone its discussion to Section 10.2.

```
(defun-nx preconditions-in-system-level-mode (x86)
  (and
   ;; The x86 state is well-formed.
   (x86p x86)
   ;; The model is operating in the system-level mode.
   (system-level-mode x86)
```

```
(page-structure-marking-mode x86)
;; The program is located at linear addresses ranging from
;; (rip x86) to (+ -1 (len *program*) (rip x86)).
(program-at (create-canonical-address-list
             (len *program*) (rip x86))
            *program* x86)
;; No error is encountered when translating the program's linear
;; addresses to physical addresses.
(not (mv-nth 0
             (las-to-pas
              (create-canonical-address-list
               (len *program*) (rip x86))
              :x (cpl x86) x86)))
;; The program's physical addresses are disjoint from the
;;  physical addresses of the paging structures.
(disjoint-p
 (mv-nth 1
         (las-to-pas (create-canonical-address-list
                      (len *program*) (rip x86))
                     :x (cpl x86) x86))
 (open-qword-paddr-list
  (gather-all-paging-structure-qword-addresses x86)))
;; The addresses used in the program are canonical.
(canonical-address-p (rip x86))
(canonical-address-p (+ (len *program*) (rip x86)))
;; The initial state is error-free.
(equal (ms x86) nil)
(equal (fault x86) nil)))
```

We now symbolically simulate this program in the system-level mode of operation of our x86 ISA model. In addition to the updates to the instruction pointer and the carry flag, the conclusion of `program-effects-in-system-level-mode-2` also captures the effects of the page walks done to fetch the instructions from the linear memory.

```
(defthm program-effects-in-system-level-mode-2
  (implies
   (preconditions-in-system-level-mode x86)
   (equal (x86-run 2 x86)
          (!rip (+ 2 (xr :rip 0 x86))
                (!flgi *cf* 1
                       (mv-nth 2
                               (las-to-pas
                                (list (rip x86) (+ 1 (rip x86)))
                                :x (cpl x86) x86)))))))
```

Compare this to `program-effects-in-user-level-mode-2`, previously presented in Section 9.1.

```
(defthm program-effects-in-user-level-mode-2
  (implies (preconditions-in-user-level-mode x86)
           (equal (x86-run 2 x86)
                  (!rip (+ 2 (rip x86)) (!flgi *cf* 1 x86)))))
```

Thus, symbolic simulation in the system-level mode necessarily leads to cluttered symbolic expressions.

10.2 Sub-Modes of the System-level Mode

We offer two sub-modes of operation of the system-level mode — the *marking mode* and the *non-marking mode*. If operation in the marking mode is desired, the field `page-structure-marking-mode` in the x86 state is set to `t` and to `nil` otherwise. It is in the marking mode of operation that the system-level mode is the true specification of the x86 ISA. In the previous section, we described the symbolic simulation of our example program in the system-level marking mode. In the non-marking mode, updates to `A` and `D` flags are ignored (that is, the paging structures

are not marked during traversals), which simplifies theorems that drive our symbolic simulation framework.

Before discussing the complications stemming from updates to the `A` and `D` flags that motivated the provision of these separate modes, we present the `*program*` example in the system-level non-marking mode.

```
(defun-nx preconditions-in-non-marking-mode (x86)
  (and
   ;; The x86 state is well-formed.
   (x86p x86)
   ;; The model is operating in the system-level non-marking mode.
   (system-level-mode x86)
   (not (page-structure-marking-mode x86))
   ;; The program is located at linear addresses ranging from
   ;;  (rip x86) to (+ -1 (len *program*) (rip x86)).
   (program-at (create-canonical-address-list
                (len *program*) (rip x86))
               *program* x86)
   ;; No error encountered when translating the program's linear
   ;; addresses to physical addresses.
   (not (mv-nth 0
                (las-to-pas
                 (create-canonical-address-list
                  (len *program*) (rip x86))
                 :x (cpl x86) x86)))
   ;; The addresses used in the program are canonical.
   (canonical-address-p (rip x86))
   (canonical-address-p (+ (len *program*) (rip x86)))
   ;; The initial state is error-free.
   (equal (ms x86) nil)
   (equal (fault x86) nil)))
```

Observe that function `preconditions-in-system-level-mode` discussed in

the previous section is largely similar to `preconditions-in-non-marking-mode`, except for the following crucial differences:

1. The predicate `(page-structure-marking-mode x86)` evaluates to true in the former and to false in the latter function.

2. Unlike the former, the latter function does not include the condition that the program's physical addresses should be disjoint from those of the paging data structures.

This disjointness condition of the program and the paging structures is not required in the non-marking mode because `A` and `D` flags are not updated during paging structure traversals made on behalf of instruction fetches (indeed, this is true for all paging structure traversals, irrespective of their origin, in the non-marking mode). This means that even if the program and the paging structures are located at the same physical memory addresses (which is unlikely for "reasonable" programs), the program will not be modified over its course of execution. Of course, unlike in our example, if a program actively modified some paging entries (for example, to map a new page or to remove a page mapping), then we would require this disjointness condition to be a part of the preconditions for symbolic simulation regardless of whether we are in marking mode.

The conclusion of `program-effects-in-non-marking-mode-2` is the same as that of `program-effects-in-user-level-mode-2`, though, of course, the preconditions are different.

```
(defthm program-effects-in-non-marking-mode-2
  (implies (preconditions-in-non-marking-mode x86)
           (equal (x86-run 2 x86)
                  (!rip (+ 2 (rip x86)) (!flgi *cf* 1 x86)))))
```

Thus, the system-level non-marking mode gives us uncluttered symbolic expressions, similar to those we get in the user-level mode.

10.2.1 Linear Memory Read-over-Write Theorems

The absence of the disjointness condition of the program and the paging structures in `preconditions-in-non-marking-mode` is really because of the simpler non-interference theorems in the non-marking mode. Specifically, the side-effect updates to the `A` and `D` flags complicate Read-over-Write theorems about linear memory in the marking mode of operation.

Consider the linear memory non-interference theorem in the system-level marking mode. Let a linear memory read operation access linear addresses `las-1`, whose corresponding physical addresses are `pas-1`. Let a linear memory write operation update linear addresses `las-2`, whose corresponding physical addresses are `pas-2`. Then, the write operation will not interfere with the read operation (i.e., the read operation will return the same value, irrespective of whether the write occurred or not) if the following conditions are true:

1. Physical addresses `pas-1` and `pas-2` are distinct.

2. Physical addresses `pas-1` are disjoint from all the physical addresses of the translation-governing entries of `las-2`; i.e., the read operation does not read

the paging entries pertaining to the translation of `las-2`.

3. Physical addresses `pas-1` are disjoint from all the physical addresses of the translation-governing entries of `las-1`; i.e., the read operation does not read the paging entries pertaining to the translation of `las-1`.

4. Physical addresses `pas-2` are disjoint from all the physical addresses of the translation-governing entries of `las-1`; i.e., the write operation does not modify the paging entries pertaining to the translation of `las-1`.

The statement of this theorem in ACL2 is as follows.

```
(defthm rb-wb-disjoint-in-system-level-mode
  (let* ((pas-1 (mv-nth 1 (las-to-pas las-1 r-x (cpl x86) x86)))
         (las-2 (strip-cars addr-lst))
         (pas-2 (mv-nth 1 (las-to-pas las-2 :w (cpl x86) x86))))
    (implies
     (and
      ;; Condition 1
      (disjoint-p pas-1 pas-2)
      ;; Condition 2
      (disjoint-p pas-1
                  (all-xlation-governing-entries-paddrs las-2 x86))
      ;; Condition 3
      (disjoint-p pas-1
                  (all-xlation-governing-entries-paddrs las-1 x86))
      ;; Condition 4
      (disjoint-p pas-2
                  (all-xlation-governing-entries-paddrs las-1 x86))
      (canonical-address-listp las-1)
      (addr-byte-alistp addr-lst)
      (system-level-mode x86)
      (x86p x86))
     (and
```

```
;; Error flags of rb are equal.
(equal (mv-nth 0 (rb las-1 r-x (mv-nth 1 (wb addr-lst x86))))
       (mv-nth 0 (rb las-1 r-x x86)))
;; Bytes read by rb are equal.
(equal (mv-nth 1 (rb las-1 r-x (mv-nth 1 (wb addr-lst x86))))
       (mv-nth 1 (rb las-1 r-x x86)))))))
```

Condition 1 is an obvious non-interference criterion. Conditions 2 and 3 preclude reads from those regions of the memory that are indirectly changed by the write and read operations. Condition 4 prevents the write operation from modifying the mapping of the linear addresses from where data is read — this ensures that the same physical addresses are accessed by the read operation, irrespective of the occurrence of the write operation. One can refine these conditions to make them less restrictive. For example, the second condition may be changed to say that the read operation should not read that *portion* of the paging entries that includes the `A` and `D` flags. For simplicity, we present these conditions as is.

Observe that conditions 2 and 3 will hold in the common case of reasoning about interactions of data reads and writes by a program — that is, if `las-1` and `las-2` are both linear addresses of the program's data. This is because a program's data is usually separated from the system data structures. However, reasoning about every memory read and write interaction will require relieving the conditions presented above. This means that the theorem prover will need to work in order to rewrite each of these conditions to true. In the process of doing so, other lemmas will be used, which may have hypotheses of their own that need to be relieved. In short, the process of rewriting all these conditions can slow down symbolic simulation, thereby affecting reasoning efficiency. A theorem is always "stronger" if it has fewer

hypotheses, and it has the added advantage of being efficient to reason with in a mechanical theorem prover like ACL2. Also, a large part of program verification is discovering the preconditions under which the program behaves as expected, and these hypotheses can make the discovery of interesting or non-obvious preconditions even more arduous.

The linear memory Read-over-Write theorem in the non-marking mode does not include the conditions 2 and 3, thereby optimizing reasoning for the common case. Thus, following the same rationale as that for having separate modes for application and system program analysis, our framework allows the user to choose to operate either in the marking or non-marking mode of operation of the system-level mode.

The benefits of having these two sub-modes of the system-level mode become more apparent when reasoning about a real-world supervisor program. Over the course of verification of the *zero-copy* program for one of our case studies, we found that discovering interesting preconditions was easier in the non-marking mode. Consequently, we first verified that program in the non-marking mode and then ported this proof over to the marking mode relatively easily. Details about this proof are presented in Chapter 12.

10.2.2 Reducing Reasoning Overhead in the Marking Mode

The non-marking mode offers simplified reasoning due to the suppression of `A` and `D` flag updates, but we still have to account for them in the marking mode of operation. However, for all instructions of a program that do not modify the paging entries actively (i.e., the virtual memory abstraction remains intact) and that do

not rely on the `A` and `D` flags for any functionality, it would be desirable if these updates did not impede reasoning in the marking mode as well. In this section, we illustrate how we reduce the reasoning overhead in the marking mode using the non-interference theorem for linear memory reads as an example.

Consider the following symbolic expression, which denotes the return value of a read operation from an x86 state obtained after another read operation. Such an expression occurs frequently — for example, for successive instruction fetches. Let the physical addresses corresponding to the linear addresses `las-1` be disjoint from the physical addresses of the paging structures.

```
(mv-nth 1 (rb las-1 r-x-1
              (mv-nth 2 (rb las-2 r-x-2 x86))))
```

Note that the inner read operation modifies the x86 state by setting the `A` flags in the paging entries pertaining to linear addresses `las-2`. Thus, this expression is equal to the following one:

```
;; rb-after-page-walks:
(mv-nth 1 (rb las-1 r-x-1
              (mv-nth 2 (las-to-pas las-2 r-x-2 (cpl x86) x86))))
```

We label the expression above `rb-after-page-walks` for future reference.

Given that `las-1` are disjoint from the paging structures, `rb-after-page-walks` should be equal to the following expression:

```
(mv-nth 1 (rb las-1 r-x-1 x86))
```

That is, we should be able to infer that the value returned by the outer read is unaffected by the inner read. In order to prove this non-interference theorem for linear memory reads, it is necessary to prove that the address mapping of `las-1` is unaffected by the paging walk that translates `las-2`. More generally, we need to prove the independence of page walks.

Proving the Independence of Page Walks

Page walks in the x86 ISA model are specified by the function `las-to-pas`, which attempts to translate a list of linear addresses to their corresponding physical addresses by traversing the paging data structures. The function `las-to-pas` recursively calls another function `la-to-pa`, which attempts to translate one linear address at a time by traversing the hierarchical paging structures in order and reading a paging entry from each of those structures. Assuming there are no faults at any step, the function `la-to-pa` calls:

- Function `la-to-pa-pml4-table`, which reads a `PML4TE` and then calls...
    - Function `la-to-pa-page-dir-ptr-table`, which reads a `PDPTE` to either map a 1GB page or to call...
        - Function `la-to-pa-page-directory`, which reads a `PDE` to either map a 2MB page or to call...
            - Function `la-to-pa-page-table`, which reads a `PTE` to map a 4KB page.

The hierarchy of these functions corresponds to the hierarchy of the paging data structures — see Figures 6.3, 6.4, and 6.5 for all possible configurations of the paging data structures.

The following expressions state the independence of page walks in terms of `las-to-pas`:

```
;; Error flags of las-to-pas are equal.
(equal
 (mv-nth 0 (las-to-pas las-1 r-x-1 cpl-1
                       (mv-nth 2 (las-to-pas las-2 r-x-2 cpl-2 x86))))
 (mv-nth 0 (las-to-pas las-1 r-x-1 cpl-1 x86)))
```

and

```
;; Physical addresses returned by las-to-pas are equal.
(equal
 (mv-nth 1 (las-to-pas las-1 r-x-1 cpl-1
                       (mv-nth 2 (las-to-pas las-2 r-x-2 cpl-2 x86))))
 (mv-nth 1 (las-to-pas las-1 r-x-1 cpl-1 x86)))
```

Clearly, both the error flags and physical addresses returned by `las-to-pas` depend upon *all* the paging entries walked for address translation. For example, if a `PML4TE` is well-formed but its inferior `PDPTE` is not, then the flag of `la-to-pa-pml4-table` will signal an error.

A Naïve Approach Proving the independence of page walks would require proving the non-interference of each of the hierarchical functions, i.e., proving that partial page walks are independent. A reasonable point to start this proof is from the bottom-up, proceeding with the most inferior function to the most superior one. We begin by proving the non-interference for the error flags of `la-to-pa-page-table`.

```
;; Error flags of la-to-pa-page-table are equal.
(equal
 (mv-nth 0 (la-to-pa-page-table
            las-1 <other-arguments>
            (mv-nth 2 (la-to-pa-page-table
                       las-2 <other-arguments> x86))))
 (mv-nth 0 (la-to-pa-page-table
            las-1 <other-arguments> x86)))
```

Let `PT-1` and `PT-2` be the page table entries corresponding to the outer and inner page table walks, respectively. This proof is straightforward, because it involves considering the following two cases:

1. `PT-1` and `PT-2` are the same entry. The inner page table walk modifies only the `A` and/or `D` flags, which are not used to compute the error flags of `la-to-pa-page-table`. Therefore, the outer page table walk is unaffected.

2. `PT-1` and `PT-2` are distinct. The inner page table walk modifies a memory location disjoint from the one used for the outer page table walk. Again, the outer page table walk is unaffected.

Note that we do not have to consider the cases when `PT-1` and `PT-2` overlap because all paging entries are quadword-aligned; that is, their memory address must be a multiple of 8.

However, the problem with using this approach to prove similar theorems about functions higher up in the hierarchy is that it will require considerable case analysis. Consider the non-interference proof for the error flags involving the function `la-to-pa-page-directory`. Let `PD-1` and `PD-2` be the page directory entries

corresponding to the outer and inner page directory walks, respectively. Now we need to consider the following cases:

1. Both `PD-1` and `PD-2` map 2MB pages. The proof will proceed in a manner similar to the non-interference proof for `la-to-pa-page-table`.

2. `PD-1` maps a 2MB page, and `PD-2` refers to a `PT-2`. This case has 8 sub-cases[1], most of which cannot occur (though we still have to prove this fact). For example, `PD-1` and `PD-2` are necessarily distinct because they have different values in their `PS` fields, which indicates whether the entry maps a page or refers to another entry. We list three interesting sub-cases below.

   (a) `PD-1` and `PD-2` are distinct, but `PD-1` and `PT-2` are the same. This reduces to proving that the updates to `A/D` flags do not affect the error flag.

   (b) `PD-1` and `PD-2` are distinct, but `PD-2` and `PT-2` are the same[2]. This case is straightforward because the outer page directory walk is unaffected by entries not in its traversal path.

   (c) All the three entries — `PD-1`, `PD-2`, and `PT-2` — are distinct. This case is straightforward too because the outer page directory walk is unaffected.

3. `PD-1` refers to a `PT-1`, and `PD-2` maps a 2MB page. This case is analogous to the case above.

[1]Think of a truth table with the following three columns: (`PD-1` == `PD-2`), (`PD-1` == `PT-2`), and (`PD-2` == `PT-2`). The number of cases to consider are 2^3.

[2]The x86 ISA allows self-referencing entries — the same entry might be used as a `PML4TE`, `PDPTE`, `PDE`, and `PTE`.

4. `PD-1` refers to a `PT-1`, and `PD-2` also refers to a `PT-2`. This case has 2^6 sub-cases, which we do not list here. These sub-cases can be proved using similar arguments presented for the cases above.

The number of cases will increase as we go higher in the hierarchy because the number of paging entries to consider will increase. In addition to a burgeoning number of cases, this approach involves duplication of effort — it prevents the use of the non-interference theorem of an inferior function to prove the non-interference of a superior function. Adopting this approach essentially means that we will need to prove the non-interference theorem of a function with *every* other function in the hierarchy.

Congruence-based Rewriting Approach Recall that this whole exercise of proving the independence of page walks is a consequence of updates made to `A` and `D` flags during paging structure traversals. Since these flags only modify the x86 memory (and hence, the x86 state) but do not affect the error flag and physical address resulting from a page walk, our alternative approach involves defining the notion of a *translation-equivalent memory* as an equivalence relation, `xlate-equiv-memory`. Two x86 states satisfy this equivalence relation `xlate-equiv-memory` iff:

- both or neither specify that the x86 ISA model is in the system-level marking mode of operation,
- their paging data structures are located at the same physical memory addresses,
- the entries in the paging data structures are equal, modulo the `A` and `D` flags, and

- the rest of the physical memory is exactly equal.

Our objective is to prove two main properties:

1. The x86 state resulting from a page walk is equivalent, as dictated by the relation `xlate-equiv-memory`, to the initial x86 state.

2. Given two x86 states satisfying `xlate-equiv-memory`, a page walk performed in one state will return the same results when performed in the other state.

The independence of page walks follows immediately from the above two properties.

We now describe the proof of these two properties. We proved these properties about `la-to-pa` (and then `las-to-pas`) by proving it for each of the hierarchical functions. However, this approach requires considerably less case analysis than the previous approach because we only need to consider the effects of traversing one paging entry at a time. We illustrate this point below.

We begin by proving the first property about the lowest function in the hierarchy, `la-to-pa-page-table`, which is straightforward (given that `xlate-equiv-memory` ignores `A` and `D` flag updates) because the only effect this function has on the physical memory, if any, is the update to `A` and/or `D` flags of the relevant `PTE`. Then, we prove this property about the next function in the hierarchy, `la-to-pa-page-directory`, which has just two cases:

1. `PDE` refers to a `PTE`. We have already proved that the state returned by `la-to-pa-page-table` is equivalent to the initial state. All we have to prove is that if

the A flag of the PDE is updated, then xlate-equiv-memory is preserved. This proof is straightforward, given the definition of xlate-equiv-memory.

2. PDE maps a 2MB page. This proof proceeds in the same manner as that for la-to-pa-page-table.

Note that for functions higher in the hierarchy, the number of cases to consider will not increase. This is because this approach allows using the theorem about an inferior function to aid in the proof of the theorem about a superior function.

The proof of the second property follows the same strategy. All that is required is to prove that the A and D flags are never used by any of these hierarchical functions for referencing the next paging entry or mapping a page.

We state these two properties in ACL2 as follows.

```
;; First Property:
(defthm xlate-equiv-memory-with-mv-nth-2-las-to-pas
  (xlate-equiv-memory (mv-nth 2 (las-to-pas las r-w-x cpl x86))
                      x86)
  :rule-classes :rewrite)

;; Second Property:
(defthm xlate-equiv-memory-and-mv-nth-0-las-to-pas-cong
  (implies (xlate-equiv-memory x86-1 x86-2)
           (equal (mv-nth 0 (las-to-pas las r-w-x cpl x86-1))
                  (mv-nth 0 (las-to-pas las r-w-x cpl x86-2))))
  :rule-classes :congruence)

(defthm xlate-equiv-memory-and-mv-nth-1-las-to-pas-cong
  (implies (xlate-equiv-memory x86-1 x86-2)
           (equal (mv-nth 1 (las-to-pas las r-w-x cpl x86-1))
```

```
                (mv-nth 1 (las-to-pas las r-w-x cpl x86-2))))
  :rule-classes :congruence)
```

Note that the two theorems of the second property are stored as ACL2 congruence rules [11, 67, 130]. Congruence rules allow rewriting with general equivalence relations instead of just `equal` or `iff` relations. The congruence rules state that the equality of the error flags and physical addresses returned by `las-to-pas` holds if `xlate-equiv-memory` holds for their last arguments and the other arguments are unchanged. These congruence rules allow ACL2 to *switch the context* from `equal` to `xlate-equiv-memory`. The rewrite rule of the first property enables ACL2 to rewrite the state resulting from the inner page walk to `x86` if the context is `xlate-equiv-memory`.

```
;; Error flags of las-to-pas are equal.
(mv-nth 0 (las-to-pas las-1 r-x-1 cpl-1
                      (mv-nth 2 (las-to-pas las-2 r-x-2 cpl-2 x86))))
==
;; xlate-equiv-memory-and-mv-nth-0-las-to-pas-cong switches
;; the context from equal to xlate-equiv-memory for the
;; last argument of the outer function call.
;; xlate-equiv-memory-with-mv-nth-2-las-to-pas is now
;; applicable at the last argument of the outer function call.
(mv-nth 0 (las-to-pas las-1 r-x-1 cpl-1 x86))

;; Physical addresses returned by las-to-pas are equal.
(mv-nth 1 (las-to-pas las-1 r-x-1 cpl-1
                      (mv-nth 2 (las-to-pas las-2 r-x-2 cpl-2 x86))))
==
;; xlate-equiv-memory-and-mv-nth-1-las-to-pas-cong switches
;; the context from equal to xlate-equiv-memory for the
;; last argument of the outer function call.
```

```
;; xlate-equiv-memory-with-mv-nth-2-las-to-pas is now
;; applicable at the last argument of the outer function call.
(mv-nth 1 (las-to-pas las-1 r-x-1 cpl-1 x86))
```

Using these rules, ACL2 can *automatically* prove the independence of page walks.

Conditional Congruence-based Rewriting

Given the independence of page walks, we can now prove the following theorem in ACL2.

```
(defthm rb-and-xlate-equiv-memory-disjoint-from-paging-structures
  (implies (and
            ;; Hypotheses binding x86-2 to a suitable term elided.
            (xlate-equiv-memory x86-1 x86-2)
            (disjoint-p
             (mv-nth 1 (las-to-pas las r-x (cpl x86-1) x86-1))
             (open-qword-paddr-list
              (gather-all-paging-structure-qword-addresses x86-1)))
            (canonical-address-listp las))
           (equal (mv-nth 1 (rb las r-x x86-1))
                  (mv-nth 1 (rb las r-x x86-2))))
  :rule-classes :rewrite)
```

Note that this theorem can aid in rewriting `rb-after-page-walks` discussed at the very beginning of this section — that is, the value returned by `rb` is unaffected by a previous `rb`, as long as the former `rb` reads from physical addresses that are disjoint from the physical addresses containing the paging data structures.

```
(equal
 ;; rb-after-page-walks:
```

```
(mv-nth 1 (rb las-1 r-x-1
              (mv-nth 2 (las-to-pas las-2 r-x-2 (cpl x86) x86))))
(mv-nth 1 (rb las-1 r-x-1 x86)))
```

Theorem `rb-and-xlate-equiv-memory-disjoint-from-paging-structures` cannot be defined as a congruence rule, which can only be of the form `(implies (equiv1 ...) (equiv2 ...))`. This poses a problem when rewriting expressions like `rb-after-page-walks`. Given this rewrite rule above, `rb-after-page-walks` may not be *automatically* rewritten to `(mv-nth 1 (rb las-1 r-x-1 x86))`. This is because the rewrite rule has `x86-2` as a free variable [12], which means that ACL2 has to search for an appropriate binding for `x86-2` when trying to relieve the first hypothesis of this rule. ACL2 may or may not succeed at finding the right instantiation due to the lack of contextual information. Thus, the user may have to guide the theorem prover manually by providing hints.

We solve this problem by performing *conditional congruence-based rewriting*. We define a function called `rb-alt`, which is equal to `rb` only under certain conditions that (in part) require that the addresses it reads from are disjoint from the paging structures, and returns `nil` value otherwise. Consequently, we can prove the following congruence rule about `rb-alt`:

```
(defthm mv-nth-1-rb-alt-and-xlate-equiv-memory-cong
  (implies (xlate-equiv-memory x86-1 x86-2)
           (equal (mv-nth 1 (rb-alt las r-w-x x86-1))
                  (mv-nth 1 (rb-alt las r-w-x x86-2))))
  :rule-classes :congruence)
```

We rewrite calls of `rb` to `rb-alt` where applicable. Together, this equality-based rewriting and the congruence rule above allow ACL2 to switch the context from

equal to xlate-equiv-memory for the last argument of rb in rb-after-page-walks, thereby enabling automatic simplification to (mv-nth 1 (rb-alt las-1 r-x-1 x86)).

```
;; rb-after-page-walks:
(mv-nth 1 (rb las-1 r-x-1
              (mv-nth 2 (las-to-pas las-2 r-x-2 (cpl x86) x86))))
==
;; rb is rewritten to rb-alt because las-1 are disjoint from
;; the paging structures.
(mv-nth 1 (rb-alt las-1 r-x-1
                  (mv-nth 2 (las-to-pas las-2 r-x-2 (cpl x86) x86))))
==
;; Congruence rule mv-nth-1-rb-alt-and-xlate-equiv-memory-cong
;; allows xlate-equiv-memory-with-mv-nth-2-las-to-pas to
;; rewrite the last argument of rb-alt above to x86.
(mv-nth 1 (rb-alt las-1 r-x-1 x86))
```

Defining rb-alt does have the comparatively small drawback of needing to prove theorems about rb-alt that are similar to those previously proved about rb. However, this is a one-time task and these theorems are trivial to prove.

Our lemma libraries include rules that facilitate such automatic simplification for other functions as well, thereby reducing verification overhead due to A and D flag updates whenever possible. An example is the program-at function, which defines where a program is located in the linear memory of the x86 state. We rewrite program-at to program-at-alt if the program is known to be disjoint from the paging structures, and we prove a congruence rule about program-at-alt, à la mv-nth-1-rb-alt-and-xlate-equiv-memory-cong, that allows a context switch to xlate-equiv-memory from equal.

Remarks

Observe that instead of rewriting `rb` to `rb-alt` when applicable, we could have simply proven a rule that rewrote the expression `rb-after-page-walks` in terms of just `rb`:

```
(defthm mv-nth-1-rb-after-mv-nth-2-las-to-pas
  (implies
   (and
    (disjoint-p (mv-nth 1 (las-to-pas las-1 r-x-1 (cpl x86) x86))
                (open-qword-paddr-list
                 (gather-all-paging-structure-qword-addresses x86)))
    (canonical-address-listp las-1))
   (equal
    (mv-nth 1 (rb las-1 r-x-1
                  (mv-nth 2 (las-to-pas las-2 r-w-x-2 cpl x86))))
    (mv-nth 1 (rb las-1 r-x-1 x86))))
  :rule-classes :rewrite)
```

This rule above serves the same purpose as conditional congruence-based rewriting. However, a benefit of conditional congruence-based rewriting is that it helps in distinguishing between reads done from the paging structures and reads done from the rest of the memory — the former are in terms of `rb` and the latter are in terms of `rb-alt`. This is extremely convenient during an interactive proof session because a user can quickly understand the nature of memory access operations issued by a program. Another benefit of using congruence rules is that they are more general than `mv-nth-1-rb-after-mv-nth-2-las-to-pas`; they apply to all functions that, like `las-to-pas`, result in an x86 state that satisfies `xlate-equiv-memory` with the input x86 state.

When `rb` reads a paging entry (or any memory location within the paging structures), it cannot be rewritten to `rb-alt`, obviously. Such a read must necessarily consider the updates made to the `A` and/or `D` flags, irrespective of whether these updates are due to the marking side-effect of page walks or due to any modifications a program might make. In this case, our rules about the non-interference or overlap involving `rb` with other accessors or updaters aid in symbolic simulation. Our libraries also include lemmas that describe the x86 state after a page walk precisely (i.e., using `equal` instead of `xlate-equiv-memory`) — specifically, these lemmas state which paging addresses were modified (à la address enumeration techniques [79]) and how they were modified.

Part IV

Case Studies

Chapter 11

Analysis of User-Mode Programs

In this chapter, we illustrate our methodology to verify application programs by describing the analyses of two user-mode programs — *pop-count* and *word-count* — performed in the user-level mode of operation of our x86 ISA model. Pop-count is a straight-line program that involves complicated bit manipulation operations, and its verification effort was originally presented in our VSTTE'13 paper [167]. Word-count is a program with a loop and branches that exhibits non-determinism via system calls, and its verification effort was originally presented in our FMCAD'14 paper [170]. These two different kinds of programs demonstrate the capabilities of our x86 ISA model and reasoning framework for the verification of user-mode programs from the point of view of an application programmer.

Section 11.1 describes the pop-count proof, while Section 11.2 presents the word-count proof. We conclude this chapter with a few remarks about both these case studies in Section 11.3.

11.1 Pop-Count Program

The pop-count or population-count program computes the number of ones in the binary representation of a given input. The following two C sub-routines are

an optimized implementation of pop-count. The sub-routine `popcount_32` has been taken from Sean Anderson's bit twiddling hacks [165], and it counts the non-zero bits in its 32-bit unsigned integer input. The sub-routine `popcount_64` computes this count for its 64-bit unsigned integer input by calling `popcount_32` twice, once each for the least and most significant 32 bits.

```
int popcount_32 (unsigned int v) {
  v = v - ((v >> 1) & 0x55555555);
  v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
  v = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
  return(v);
}

int popcount_64 (long unsigned int v) {
  long unsigned int v1, v2;
  v1 = (v & 0xFFFFFFFF); /* v1: lower 32 bits of v */
  v2 = (v >> 32);        /* v2: upper 32 bits of v */
  return (popcount_32(v1) + popcount_32(v2));
}
```

On compiling a program containing these sub-routines with the GCC compiler (optimization level 2) on a Linux machine, we get the following machine/assembly code.

```
0000000000400610 <popcount_32>:
  400610:        89 fa                   mov    %edi,%edx
  400612:        d1 ea                   shr    %edx
  400614:        81 e2 55 55 55 55       and    $0x55555555,%edx
  40061a:        29 d7                   sub    %edx,%edi
  40061c:        89 f8                   mov    %edi,%eax
  40061e:        c1 ef 02                shr    $0x2,%edi
  400621:        25 33 33 33 33          and    $0x33333333,%eax
  400626:        81 e7 33 33 33 33       and    $0x33333333,%edi
  40062c:        01 c7                   add    %eax,%edi
  40062e:        89 f8                   mov    %edi,%eax
```

```
  400630:       c1 e8 04                shr    $0x4,%eax
  400633:       01 f8                   add    %edi,%eax
  400635:       25 0f 0f 0f 0f          and    $0xf0f0f0f,%eax
  40063a:       69 c0 01 01 01 01       imul   $0x1010101,%eax,%eax
  400640:       c1 e8 18                shr    $0x18,%eax
  400643:       c3                      retq
  400644:       66 66 66 2e 0f 1f 84    data32 data32
  40064b:       00 00 00 00 00          nopw %cs:0x0(%rax,%rax,1)

0000000000400650 <popcount_64>:
  400650:       89 fa                   mov    %edi,%edx
  400652:       89 d1                   mov    %edx,%ecx
  400654:       d1 e9                   shr    %ecx
  400656:       81 e1 55 55 55 55       and    $0x55555555,%ecx
  40065c:       29 ca                   sub    %ecx,%edx
  40065e:       89 d0                   mov    %edx,%eax
  400660:       c1 ea 02                shr    $0x2,%edx
  400663:       25 33 33 33 33          and    $0x33333333,%eax
  400668:       81 e2 33 33 33 33       and    $0x33333333,%edx
  40066e:       01 c2                   add    %eax,%edx
  400670:       89 d0                   mov    %edx,%eax
  400672:       c1 e8 04                shr    $0x4,%eax
  400675:       01 c2                   add    %eax,%edx
  400677:       48 89 f8                mov    %rdi,%rax
  40067a:       48 c1 e8 20             shr    $0x20,%rax
  40067e:       81 e2 0f 0f 0f 0f       and    $0xf0f0f0f,%edx
  400684:       89 c1                   mov    %eax,%ecx
  400686:       d1 e9                   shr    %ecx
  400688:       81 e1 55 55 55 55       and    $0x55555555,%ecx
  40068e:       29 c8                   sub    %ecx,%eax
  400690:       89 c1                   mov    %eax,%ecx
  400692:       c1 e8 02                shr    $0x2,%eax
  400695:       81 e1 33 33 33 33       and    $0x33333333,%ecx
  40069b:       25 33 33 33 33          and    $0x33333333,%eax
  4006a0:       01 c8                   add    %ecx,%eax
  4006a2:       89 c1                   mov    %eax,%ecx
  4006a4:       c1 e9 04                shr    $0x4,%ecx
```

```
4006a7:        01 c8                   add    %ecx,%eax
4006a9:        25 0f 0f 0f 0f          and    $0xf0f0f0f,%eax
4006ae:        69 d2 01 01 01 01       imul   $0x1010101,%edx,%edx
4006b4:        69 c0 01 01 01 01       imul   $0x1010101,%eax,%eax
4006ba:        c1 ea 18                shr    $0x18,%edx
4006bd:        c1 e8 18                shr    $0x18,%eax
4006c0:        01 d0                   add    %edx,%eax
4006c2:        c3                      retq
4006c3:        66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
4006ca:        00 00 00
4006cd:        0f 1f 00                nopl   (%rax)
```

Even though the `popcount_64` C sub-routine calls `popcount_32` twice, the machine code corresponding to `popcount_64` does not make any such calls — instead, it contains instructions from two calls of `popcount_32` interleaved together. Compilers frequently make such optimizations in the interest of execution efficiency. An implication of such optimizations is that theorems about `popcount_32` cannot be used directly for deriving proofs of correctness of `popcount_64`. Thus, we focus on proving the functional correctness of `popcount_64`.

11.1.1 `GL`: Proving Theorems about Finite Objects Automatically

Reasoning about `popcount_64` using traditional theorem proving techniques imposes a burden on the user of proving various lemmas about bit-vector arithmetic. This is a laborious task, especially since it is non-obvious how the implementation of `popcount_64` meets its specification. Instead, we use a pre-existing ACL2 library called `GL` [172, 175] to prove *automatically* the correctness of `popcount_64`.

GL is a bit-blasting procedure used to prove theorems about finite objects by using either a verified BDD engine or an external SAT solver. The key idea of GL is to encode finite ACL2 objects into symbolic objects represented by boolean expressions, thereby reducing computations done on the finite objects to those done using BDD or AIG operations. The conversion to symbolic objects is verified in ACL2, as are the BDD operations. For this program, we use GL in its BDD mode.

GL performs computations using logical ACL2 definitions — for example, the :logic part of an mbe is used for bit blasting, not the :exec part. Similarly, GL uses the logical representation of the x86 state defined by an abstract stobj, not the Lisp vector representation used for execution. How does GL deal with constrained or uninterpreted functions, such as create-undef (see Section 7.1), which do not have a conventional logical definition? We use the def-gl-rewrite feature to prove rewrite rules about such functions, thereby allowing reasoning at the term-level [34] in addition to at the usual bit-level. Normal ACL2 rewrite rules are not used by GL, but those proved using def-gl-rewrite are recognized by GL. For this program, create-undef is used only to provide undefined values for 1-bit flags. We just need two rewrite rules about create-undef. One rule says that create-undef always returns an integer, which is already a known ACL2 fact about this function and hence, is trivial to prove using def-gl-rewrite. The other rule forces a case-split on the value of the least significant bit of create-undef (which provides the 1-bit undefined value), thereby allowing the reduction of an expression involving a

constrained function to a normal GL number of width 1 bit[1].

Bit-blasting can be used to prove theorems if the following two conditions are satisfied:

1. All the free variables in the statement are finite (e.g., 64-bit integers, a fixed-length list, etc.) and have been assigned a symbolic object.

2. These symbolic objects cover all the possible cases necessary for the proof.

A command called def-gl-thm is used to prove theorems using GL. This command allows the user to state a conjecture — the hypotheses using the keyword :hyp and the conclusion using :concl — and the bindings of the free variables to symbolic objects using :g-bindings. These bindings also specify a BDD variable order, which may influence bit-blasting efficiency significantly. A conjecture submitted using def-gl-thm is proved if and only if GL bit-blasts it to an expression representing a non-nil value and the two conditions above are met — i.e., the finite symbolic objects are proved by GL to cover the domain of all the free variables in the conjecture.

11.1.2 Verification of popcount_64

Let the ACL2 constant *popcount-64* represent the x86 machine-code program shown at the beginning of this Section. The constant *popcount-64* is simply an association list mapping addresses to instruction bytes.

[1]We thank Sol Swords for helping us formulate the appropriate def-gl-rewrite rules involving create-undef.

```
(defconst *popcount-64*
  (list
   (cons #x400650 #x89) ;; mov %edi,%edx
   (cons #x400651 #xfa)
   ;; ...                ... many instructions elided ...
   (cons #x4006c2 #xc3) ;; retq
   (cons #x4006c3 #x66) ;; nopw %cs:0x0(%rax,%rax,1)
   (cons #x4006c4 #x2e)
   (cons #x4006c5 #x0f)
   (cons #x4006c6 #x1f)
   (cons #x4006c7 #x84)
   (cons #x4006c8 #x00)
   (cons #x4006c9 #x00)
   (cons #x4006ca #x00)
   (cons #x4006cb #x00)
   (cons #x4006cc #x00)
   (cons #x4006cd #x0f) ;; nopl (%rax)
   (cons #x4006ce #x1f)
   (cons #x4006cf #x00)
   ))
```

The following theorem, `x86-popcount-64-correct`, shows that the `popcount_64` sub-routine is correct. This theorem states that given an initial x86 state — where the instruction pointer points to the linear address `0x400650`, the field `user-level-mode` is `t`, the program `*popcount-64*` is located in the linear memory, and the register `*rdi*` contains the 64-bit unsigned integer input — running the x86 interpreter till the halt address `0x4006c2` is encountered will produce an x86 state whose `*rax*` register contains the population count of `n` and the instruction pointer points to the instruction following the halt address. The native Lisp (and ACL2) function `logcount` [28] is a good candidate for the specification of population count — this function computes the number of non-zero ones in a positive integer using

a straightforward recursion, and hence, is easier to comprehend than the algorithm implemented by `popcount_64`. The `b*` construct [2] below is an advanced macro that serves as a replacement for the `let*` construct.

```
(def-gl-thm x86-popcount-64-correct
  :hyp (and (natp n)
            (< n (expt 2 64)))

  :concl
  (b* ((start-address #x400650)            ;; Address of first
                                           ;; instruction
       (halt-address #x4006c2)             ;; Address of return
                                           ;; instruction
       (x86 (create-x86))
       (x86
        (!user-level-mode t x86))
       ((mv flg x86)                       ;; Program is placed in
        (init-x86-state                    ;; appropriate memory
         nil start-address halt-address    ;; locations.  MS and
         nil nil nil 0 *popcount-64* x86)) ;; FAULT fields are
                                           ;; set to nil.
                                           ;; Returns: an error
                                           ;; flag and a
                                           ;; modified state

       (x86 (wr64 *rdi* n x86))            ;; Input is in rdi.

       (x86 (x86-run 300 x86)))            ;; 300 is the upper
                                           ;; limit on the number
                                           ;; of instructions to
                                           ;; execute. If halt
                                           ;; address is
                                           ;; encountered earlier,
                                           ;; execution halts.

    (and
```

```
    (equal flg nil)                          ;; No error was
                                             ;; encountered during
                                             ;; state initialization.

    (equal (rgfi *rax* x86) (logcount n)) ;; rax contains the
                                             ;; population count of n.

    (equal (rip x86) (+ 1 halt-address)))) ;; rip contains the
                                             ;; address of the
                                             ;; instruction following
                                             ;; the halt address.

  :g-bindings                                ;; n is bound to a
                                             ;; symbolic object
                                             ;; represented with
                                             ;; 65 boolean variables:
                                             ;; 64 for the number of
                                             ;; bits in n and 1 for
                                             ;; its sign.
  `((n   (:g-number ,(gl-int 0 1 65)))))
```

The theorem `x86-popcount-64-correct` is proved completely automatically in around 90 seconds on a machine with Intel Xeon E31280 CPU @ 3.50GHz. Bit-blasting has traditionally been used for smaller problems, but its use to verify x86 machine code has made possible by two main factors:

1. *Lean Representation of the x86 State:* In a previous version of our x86 ISA model where the state's logical representation included large linear lists corresponding to the memory (discussed previously in Section 5.2.1.1), `GL` needed to create symbolic representations of these lists and manipulate them for every memory access or update. Thus, bit blasting did not scale as a proof procedure

for that version of our ISA model. Proofs using bit blasting were eventually facilitated by our use of abstract stobjs to specify the x86 state, which used a single sparse record structure to represent the memory.

2. *Industrial-Strength Capacity of* `GL`: `GL` is a "serious" tool, used regularly in the industry for hardware verification [58, 85, 87, 184]. It has a number of optimizations that enable fast BDD operations and efficient bit-blasting.

11.1.3 Counterexamples to Fix Incorrect Specifications or Programs

If `GL` fails to prove a given conjecture, then it can produce counterexamples. Counterexamples can help in debugging failed proofs and formulating correct properties. We illustrate this by introducing a bug in the machine code corresponding to `popcount_32`. Suppose the shift instruction at the linear address `0x400640` of the `popcount_32` sub-routine was replaced by `nopl (%rax)` (instruction: `0x0f 0x1f 0x00`). We submit the following event to ACL2; note that we request 2 counterexamples in the case of a failed proof.

```
(def-gl-thm x86-popcount-32-correct
  :hyp (and (natp n)
            (< n (expt 2 32)))
  :concl (b* ((start-address #x400610)
              (halt-address #x400643)
              (x86 (create-x86))
              (x86 (!user-level-mode t x86))
              ((mv flg x86)
               (init-x86-state
                nil start-address halt-address
                nil nil nil 0 *popcount-32-buggy* x86))
              (x86 (wr32 *rdi* n x86))
```

```
            (x86 (x86-run 300 x86)))
          (equal (rgfi *rax* x86) (logcount n)))
:g-bindings
`((n    (:g-number ,(gl-int 0 1 33))))
:n-counterexamples 2)
```

This proof fails and the following values of `n` are presented by `GL` as counterexamples: `0x80000000` and `0xffffffff`. We can now perform concrete runs of the program on the x86 ISA model with these two values of `n`. In the first case, the register `rax` in the final state has the value `0x1000000` and in the second case, it has the value `0x20181008`. This gives us a clue about the bug — if the lower 24 bits of `rax` in both the cases are discarded, the remaining bits give the correct population count of the input. To check whether this observation is true for all values of `n`, we submit a modified version of `x86-popcount-32-correct`, where the only difference is in the body of the `b*` in the conclusion:

```
(equal (ash (rgfi *rax* x86) -24) (logcount n))
```

This time, `x86-popcount-32-correct` succeeds. Now, a user knows enough to either fix the program or the specification. If pop-count was indeed the behavior required, then the program can be altered by modifying its return value to be that obtained by shifting the contents of the `rax` register right by 24 bits. Or, if the program cannot be altered (maybe because its behavior was being reverse-engineered using our framework or because its calling program would include a fix), then its specification can be re-formulated to be `(logcount n)` shifted left by 24 bits.

In our VSTTE'13 paper [167], we use our pop-count example to show how this capability to produce counterexamples can also help in finding corner cases, even

if there is only one in 2^{64} cases, in program implementations.

11.1.4 Observations

Observe that the only responsibilities of the user in this verification effort were to provide the program's specification and frame its statement of correctness. No expertise in constructing bit-vector arithmetic proofs, either from scratch or from judicious use of pre-existing ACL2 libraries, was required. The user did not need our general-purpose lemma libraries, discussed in Chapter 9, to optimize reasoning efficiency or control the program's symbolic simulation. The low overhead and high automation offered by this approach can be used to speed up the proof development process in an interactive theorem-proving environment by enabling compositional verification. In principle, snippets of straight-line code in a larger x86 machine-code program can be proved equivalent to simpler specification functions, and the proof of correctness of the entire program can be obtained by stitching these simpler specification functions together.

This bit-blasting approach to machine-code analysis works best for straight-line code due to scalability reasons. It should be noted that the logical definitions being bit-blasted — i.e., the x86 ISA specification functions — are big and complicated, and the use of efficient data structures to represent the x86 state can only take us so far. Moreover, GL, like all automatic tools, has a limit to its capacity. User intervention will generally be required for programs that have loops, recursion, and other complicated constructs.

11.2 Word-Count Program

The word-count program has been taken from the book "The C Programming Language" by Kernighan and Ritchie [70], with some modifications. It is a "no-frills" version of `wc` found in Unix systems.

```
#define IN   1    /* inside a word  */
#define OUT  0    /* outside a word */
#define EOF '#'   /* EOF character  */
#include <stdio.h>
int gc(void) {
  char buf1;
  int n;
  __asm__ volatile
    (
     "mov $0x0, %%rax\n\t"
     "xor %%rdi, %%rdi\n\t"
     "mov %1, %%rsi\n\t"
     "mov $0x1, %%rdx\n\t"
     "syscall"
     : "=a"(n)
     : "g"(buf)
     : "%rdi", "%rsi", "%rdx");
  return (unsigned char) buf0;
}
/* Count lines, words, characters in input */
int main () {
  int c, nl, nw, nc, state;
  state = OUT;
  nl = nw = nc = 0;
  while ((c = gc()) != EOF) {
    ++nc;
    if (c == '\n')
      ++nl;
    if (c == ' ' || c == '\n' || c == '\t')
      state = OUT;
```

```
    else if (state == OUT) {
      state = IN;
      ++nw;
    }
  }
  return 0;
}
```

Word-count reads one character at a time from `stdin`, the standard input, until the end of input is signaled by the EOF character, which is `#` (arbitrarily chosen) for this program. The character count `nc` is incremented every time a character is read. If the character is a newline, then the line count `nl` is also incremented. The word count `nw` is incremented when the `state` changes from `OUT` to `IN`.

Note that this program uses 32-bit operands, whereas modern word-count implementations use 64-bit operands. However, this program is still representative of contemporary `wc` implementations. Also, the original program from "The C Programming Language" used the `getchar` function from the standard C library `glibc` to read a character, but we chose to write our own inline assembly sub-routine `gc` instead. The primary reason for this was that the machine code corresponding to `getchar` included some instructions (e.g., AVX instructions) that are not yet supported by our x86 ISA model. The machine/assembly code of the relevant sub-routines of word-count obtained after compilation using GCC (default optimization) is as follows.

```
    <gc>:
 1:  55                    push   %rbp
 2:  48 89 e5              mov    %rsp,%rbp
 3:  53                    push   %rbx
```

```
 4:  48 8d 45 f7             lea    -0x9(%rbp),%rax
 5:  48 89 45 e0             mov    %rax,-0x20(%rbp)
 6:  48 c7 c0 00 00 00 00    mov    $0x0,%rax
 7:  48 31 ff                xor    %rdi,%rdi
 8:  48 8b 75 e0             mov    -0x20(%rbp),%rsi
 9:  48 c7 c2 01 00 00 00    mov    $0x1,%rdx
10:  0f 05                   syscall
11:  89 c3                   mov    %eax,%ebx
12:  89 5d f0                mov    %ebx,-0x10(%rbp)
13:  0f b6 45 f7             movzbl -0x9(%rbp),%eax
14:  0f b6 c0                movzbl %al,%eax
15:  5b                      pop    %rbx
16:  5d                      pop    %rbp
17:  c3                      retq

     <main>:
 1:  55                      push   %rbp
 2:  48 89 e5                mov    %rsp,%rbp
 3:  48 83 ec 20             sub    $0x20,%rsp
 4:  c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
 5:  c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%rbp)
 6:  8b 45 f4                mov    -0xc(%rbp),%eax
 7:  89 45 f0                mov    %eax,-0x10(%rbp)
 8:  8b 45 f0                mov    -0x10(%rbp),%eax
 9:  89 45 ec                mov    %eax,-0x14(%rbp)
10:  eb 3a                   jmp    <main+0x5e>
11:  83 45 f4 01             addl   $0x1,-0xc(%rbp)
12:  83 7d fc 0a             cmpl   $0xa,-0x4(%rbp)
13:  75 04                   jne    <main+0x32>
14:  83 45 ec 01             addl   $0x1,-0x14(%rbp)
15:  83 7d fc 20             cmpl   $0x20,-0x4(%rbp)
16:  74 0c                   je     <main+0x44>
17:  83 7d fc 0a             cmpl   $0xa,-0x4(%rbp)
18:  74 06                   je     <main+0x44>
19:  83 7d fc 09             cmpl   $0x9,-0x4(%rbp)
20:  75 09                   jne    <main+0x4d>
21:  c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
```

```
22:  eb 11                   jmp    <main+0x5e>
23:  83 7d f8 00             cmpl   $0x0,-0x8(%rbp)
24:  75 0b                   jne    <main+0x5e>
25:  c7 45 f8 01 00 00 00    movl   $0x1,-0x8(%rbp)
26:  83 45 f0 01             addl   $0x1,-0x10(%rbp)
27:  e8 6a ff ff ff          callq  <gc>
28:  89 45 fc                mov    %eax,-0x4(%rbp)
29:  83 7d fc 23             cmpl   $0x23,-0x4(%rbp)
30:  75 b8                   jne    <main+0x24>
31:  8b 45 f4                mov    -0xc(%rbp),%eax
32:  8b 5d f0                mov    -0x10(%rbp),%ebx
33:  8b 4d ec                mov    -0x14(%rbp),%ecx
34:  b8 00 00 00 00          mov    $0x0,%eax
35:  c9                      leaveq
36:  c3                      retq
```

This machine-code program is structurally similar to the source program. Instructions at lines `1` to `10` in the `main` sub-routine initialize memory locations corresponding to `nc`, `nl`, `nw`, and `state` with zeros, and transfer control to the beginning of the loop. The loop begins at line number `27` with a call to `gc` and it exits if the character read in by `gc` is `#` — the comparison is done by the `cmpl` instruction at line number `29`. If `#` is not encountered, then the loop continues execution after a jump to the instruction at line number `11`, from where onwards some branching is done to compare the character to a newline, space, and tab to modify the value of the counters and `state` in the stack.

The sub-routine `gc` invokes the `syscall` instruction with `0` in the `rax`, `0` in the `rdi`, address of a memory buffer in `rsi`, and `1` — the number of bytes to be read — in `rdx`. This corresponds to a `read` system call on Linux machines that reads one byte from the standard input (whose usual identifier is `0`) and stores the result

in the memory buffer. As discussed previously in Section 7.2, the specification of the `syscall` instruction in the user-level mode of the x86 ISA model is extended to provide the semantics of some common system calls. This specification makes use of the `os-info` and `env` fields in the x86 state — `os-info` specifies the operating system under consideration because system call implementations differ among different systems, and `env` provides an external environment — a subset of the file system and an oracle — that influences or is influenced by system calls.

Before embarking on the formal analysis of this machine-code program, we ran some concrete simulations on our x86 model and compared it against a real x86 machine. We found that the program behaved as expected during those simulations. This step allows bug triaging for both the program and the ISA model — finding easy-to-manifest bugs using testing can be less effort-intensive than finding them during formal analysis.

11.2.1 Verification of Word-Count

We analyze the word-count machine-code program using a traditional theorem-proving technique called the *Boyer-Moore clock functions approach* [163, 188]. This approach involves defining a clock function that describes the number of steps to be executed for the program to reach a desired state (e.g., to the final state or to the end of a particular sub-routine, etc.). The correctness of an x86 machine-code program is stated as follows: given an x86 state $\texttt{x86}_\text{i}$ that satisfies some preconditions, the final state $\texttt{x86}_\text{f}$ `= x86-run(n,` $\texttt{x86}_\text{i}$`)`, where `n` denotes the (possibly symbolic) value computed by the clock function, satisfies some specified postconditions.

We now outline our formal analysis of the word-count program. Let us first focus on the postconditions for this program. We define three simple specification functions that compute the character, line, and word counts of an input string. Our postconditions state that at the end of execution of the word-count program, the appropriate locations in the stack contain values computed by these three functions, given the string of characters obtained from the `stdin` as input. We present the simplest of these three functions, `nc-spec`, below. The `offset` argument corresponds to the index of the next character to be read from `str-bytes`, which is the string of characters obtained from stdin including the terminating character `*eof* = #`, and `nc` is the counter to keep track of the number of characters encountered; it is incremented modulo 32, i.e., wrap-around modular arithmetic is performed. The specification function `get-char` corresponds to the `gc` sub-routine.

```
(defun nc-spec (offset str-bytes nc)
  ;; Measure declaration (for termination) omitted here.
  (if (and (eof-terminatedp str-bytes)
           (natp offset)
           (< offset (len str-bytes)))
      (b* ((c (get-char offset str-bytes))
           ((when (equal c *eof*)) nc)
           (new-nc (loghead 32 (+ 1 nc))))
        (nc-spec (1+ offset) str-bytes new-nc))
    nc))
```

We verify this program by decomposing it into two parts — the part of `main` before the loop, and the loop. Let us now consider the loop, whose preconditions are as follows:

- The x86 state is well-formed.

- The program is at symbolic addresses `addr` to `addr + program-length - 1`.

- The instruction pointer `rip` points to the first instruction of the loop.

- The initial stack pointer `rsp` points to an appropriate memory location, so that the stack does not overlap with the program in any execution.

- The `env` field is well-formed. It includes a model of the `stdin` in its file system — the file descriptor corresponding to `stdin` is `0`, as assumed by the program. Also, the contents of the `stdin` file end in the `#` character, because the program does not terminate until this character is encountered.

The loop's clock function, `loop-clk`, closely follows the structure of the loop.

```
(defun loop-clk (word-state offset str-bytes)
  (if (and (eof-terminatedp str-bytes)
           (< offset (len str-bytes))
           (natp offset))

      (let ((char (get-char offset str-bytes)))
        (if (equal char *eof*)
            (gc-clk-eof)
          (b* (((mv word-state loop-steps)
                (case char
                  (*newline* (mv *out* (gc-clk-newline)))
                  (*space*   (mv *out* (gc-clk-space)))
                  (*tab*     (mv *out* (gc-clk-tab)))
                  (t
                   (if (equal word-state *out*)
                       (mv *in* (gc-clk-otherwise-out))
                     (mv word-state (gc-clk-otherwise-in)))))))
            (clk+ loop-steps
                  (loop-clk word-state (1+ offset) str-bytes)))))
    0))
```

The `word-state` argument of `loop-clk` corresponds to the `state` variable in the C program and the other two arguments mean the same as the arguments of `nc-spec`. The nullary functions `gc-clk-newline`, `gc-clk-space`, `gc-clk-tab`, `gc-clk-otherwise-out`, and `gc-clk-otherwise-in` define the number of instructions needed for an iteration of the loop, depending on which character is read by `get-char`. Thus, `loop-clk` keeps adding the number of instructions to be executed in each iteration of the loop until the `*eof*` character is encountered.

The loop correctness theorem pertaining to counting the number of characters is given by `loop-behavior-nc` below. The functions `offset` and `input` read the offset and contents of `stdin` from the `env` field (respectively). The functions `word-state` and `nc` read the appropriate stack locations for the values of the `state` variable and the character counter. This theorem states that given an x86 state that satisfies the loop preconditions, the character count stored in the memory of the state obtained after the loop runs to completion can be described by the `nc-spec` function.

```
(defthm loop-behavior-nc
  (implies
   (and (loop-preconditions addr x86)
        (equal offset (offset x86))
        (equal str-bytes (input x86))
        (equal initial-word-state (word-state x86))
        (equal initial-nc (nc x86)))
   (equal
    (nc (x86-run (loop-clk initial-word-state offset str-bytes) x86))
    (nc-spec offset str-bytes initial-nc))))
```

The proof of this theorem is obtained by strong induction on the value computed

by `loop-clk`. The base case is when this value is `0` — that happens only when the assumptions about `env` do not hold, in which case the loop preconditions are also false, thereby proving this case. For other values computed by `loop-clk`, the proof proceeds by case analysis on the character read from `stdin`. We consider the case when a newline character is encountered; let us expand both the LHS and RHS of the conclusion of `loop-behavior-nc`.

```
;; LHS:
(nc (x86-run (loop-clk initial-word-state offset str-bytes) x86))
;; Expanding the definition of loop-clk
==
(nc (x86-run
     (clk+ (gc-clk-newline)
           (loop-clk word-state (1+ offset) str-bytes))
     x86))
;; Using the run function sequential composition rule
==
(nc (x86-run
     (loop-clk word-state (1+ offset) str-bytes)
     (x86-run (gc-clk-newline) x86)))

;; RHS:
(nc-spec offset str-bytes initial-nc)
;; Expanding the definition of nc-spec
==
(nc-spec (1+ offset) str-bytes (loghead 32 (+ 1 initial-nc)))
```

The inductive hypothesis allows us to make the following assumption:

```
(let ((x86 (x86-run (gc-clk-newline) x86)))
  (implies (and (loop-preconditions addr x86)
                (equal offset (offset x86))
                (equal str-bytes (input x86))
                (equal initial-word-state (word-state x86))
```

```
        (equal initial-nc (nc x86)))
    (equal
     (nc (x86-run (loop-clk word-state offset str-bytes) x86))
     (nc-spec offset str-bytes initial-nc))))
```

We can prove `loop-behavior-nc` for the case when newline is encountered if we prove the following lemma, which allows the expanded version of `loop-behavior-nc` to match the assumption above.

```
(defthm loop-behavior-nc-lemma
  (implies
   (loop-preconditions addr x86)
   (and
    (loop-preconditions addr (x86-run (gc-clk-newline) x86))
    (equal (offset (x86-run (gc-clk-newline) x86))
           (+ 1 (offset x86)))
    (equal (input (x86-run (gc-clk-newline) x86))
           (input x86))
    (equal (word-state (x86-run (gc-clk-newline) x86))
           (word-state x86))
    (equal (nc (x86-run (gc-clk-newline) x86))
           (loghead 32 (+ 1 (nc x86)))))))
```

The proof of `loop-behavior-nc-lemma` is straightforward. It simply requires symbolically simulating the program for `(gc-clk-newline)` steps (which is a concrete number) and projecting out the relevant components from the resulting state. The proof of other cases of `loop-behavior-nc` is analogous to this case.

Let us now consider the part of the `main` before the loop. We prove the following key lemma.

```
(defthm program-behavior-nc-lemma
  (implies
```

```
(preconditions addr x86)
(and
 (loop-preconditions addr (x86-run (gc-clk-main-before-call) x86))
 (equal (offset (x86-run (gc-clk-main-before-call) x86))
        0)
 (equal (input (x86-run (gc-clk-main-before-call) x86))
        (input x86))
 (equal (word-state (x86-run (gc-clk-main-before-call) x86))
        *out*)
 (equal (nc (x86-run (gc-clk-main-before-call) x86))
        0))))
```

The `preconditions` are very similar to `loop-preconditions`, except for the value of the instruction pointer `rip`, which contains the address of the first instruction of the `main` sub-routine instead of that of the loop. Again, like `loop-behavior-nc-lemma`, this proof simply requires symbolically simulating the program for `(gc-clk-main-before-call)` steps and projecting out the relevant state components.

Given this lemma, we can prove the final theorem of correctness of character count for this program.

```
(defthm program-behavior-nc
  (implies
   (and (preconditions addr x86)
        (equal offset (offset x86))
        (equal str-bytes (input x86)))
   (equal (nc (x86-run (clock str-bytes x86) x86))
          (nc-spec offset str-bytes))))
```

where

```
(clock str-bytes x86) = (clk+ (gc-clk-main-before-call)
                              (loop-clk *out* 0 str-bytes))
```

The proofs of theorems about the word and line count are analogous.

In addition to proving the functional correctness of word-count, we also proved an important property about the state of the memory after this program runs to completion — the values in all memory locations, except the program's stack, in the final state are exactly the same as that in the initial state. This theorem provides the assurance that the word-count program did not change any values in unintended regions of memory. This theorem was proved using a similar approach as the functional correctness theorems — strong induction to obtain this guarantee about the loop and then compositional reasoning to obtain it for the entire program.

11.2.2 Observations

Unlike the pop-count program, the formal analysis of the word-count program relied heavily on our lemma libraries for managing symbolic simulation. This formal analysis also helped in the discovery of general lemmas (e.g., non-interference theorems, theorems about the well-formedness of the `env` field, etc.) that were added to our libraries for re-use in other verification efforts. Many preconditions for the correctness of this program were discovered by employing the strategy described in Chapter 9. It is worth emphasizing that this proof was done using large and complicated specification functions that model the x86 ISA.

11.3 Remarks

There are three key differences between the pop-count and word-count proofs.

1. The pop-count program has structurally simple straight-line code, and its proof was done completely automatically. The word-count program has code with a loop, branches, and a system call, and its proof required user intervention that included compositional reasoning.

2. The final theorem of correctness of pop-count is in terms of the program being located at fixed addresses that were generated by the compiler tool chain. This is because parameterizing the program addresses would stress the bit-blasting framework. However, for this program, this does not pose any hardship. If re-compiling the same program results in machine code that is located at a different memory region, one can simply re-submit the theorem's statement to ACL2 to obtain the same guarantees about the program. This will not cause any user overhead because the proof is done completely automatically using `GL`. On the other hand, because the word-count proof involved a comparatively higher amount of user interaction, it is desirable to prove a general correctness theorem that accounts for such an issue. Therefore, the correctness theorem of word-count allows the program to be position-independent.

3. The non-relevant components of the initial x86 state have been assigned concrete values for the pop-count proof — note the `(create-x86)` call in the theorem `x86-popcount-64-correct`, which assigns default initial values to the state components. The only symbolic value in the initial x86 state is that of the `rdi` register, which can contain an arbitrary 64-bit number. On the other hand, the non-relevant components of the initial x86 state for the word-count program are completely symbolic.

Some more remarks about the third point above regarding the pop-count program: a user may wish to reason about pop-count with a symbolic initial x86 state, similar to that specified for the word-count program. To this end, parts of the x86 state can be assigned symbolic values — in a manner akin to assigning a 64-bit symbolic object to the `rdi` register — and a theorem similar to `x86-popcount-64-correct`, but in terms of these additional symbolic values, can be proven correct as before. A downside of this approach is that `GL` may reach its capacity if too many state components are assigned symbolic values; this is analogous to the situation discussed in the second point above. For instance, if we use symbolic values to represent the initial contents of all the general-purpose registers, the pop-count proof takes around an hour to run to completion. Compare this to around 90 seconds for `x86-popcount-64-correct` presented in Section 11.1.

Another approach is to use our lemma libraries to obtain the effects of pop-count on a symbolic initial x86 state, similar to our word-count case study. One can then project out the symbolic expression corresponding to the `rax` register in the final x86 state, and use `GL` to prove that this expression evaluates to the `logcount` of the initial value of the `rdi` register. This approach is almost as automatic as that presented in Section 11.1 — once the user states the preconditions, the symbolic simulation of pop-count and the projection of the `rax` register from the resulting state are done automatically, thanks to our lemma libraries. Note that this approach allows reasoning about pop-count with parameterized program addresses, i.e., the program can be position-independent. This more general theorem about pop-count can be found online [53] along with the rest of our libraries.

We draw attention to the fact that our lemma libraries enabled symbolic simulation of the pop-count program without requiring any user intervention. This highlights a contribution of this dissertation — namely, making formal verification of x86 programs using an interpreter-based approach practical. At the beginning of this dissertation project, reasoning about x86 machine code using our ISA model required significant user effort — this was unsurprising, given the complexity of the semantics of the x86 ISA. As we verified more programs using our framework, we incorporated more lemmas and proof strategies in our libraries, thereby adding functionality that can benefit future verification efforts.

These two case studies illustrate the verification strategies to adopt when using our analysis framework to reason about application programs. In general, given a computationally-intensive but structurally simple program that involves finite-sized objects, a methodology similar to the pop-count case study can be used. If the program is structurally complex, an approach similar to the word-count case study can be used. Of course, one may use `GL` to reason about computationally-intensive machine-code snippets in structurally complex programs.

Chapter 12

Analysis of a Supervisor-Mode Program

In this chapter, we illustrate how the system-level mode of operation of our x86 ISA model is used to mechanically reason about supervisor-mode programs. We describe the formal analysis of a *zero-copy* program that operates at the highest privilege level (CPL = 0) and "copies" 1GB of data. This program maps the destination and source linear addresses to the same physical addresses — it modifies a paging data entry of the destination so that the destination page is the same as the source page, where both the pages are of size 1GB. Thus, two copies of 1GB data appear to exist in the linear address space (at the source and destination) while only the original data exists in the physical address space — hence the name *zero-copy*.

All 64-bit programs — irrespective of their privilege levels — use linear addresses that are then translated to physical addresses via IA-32e paging. Since linear memory is the only view of memory available to 64-bit programs, it is critical that any update to the paging data structures should be made correctly. Supporting the verification of programs that modify system data structures is essential because they are used extensively in operating systems to perform a variety of tasks. Zero-copy programs are used to provide efficient data transfers [117, 149, 173], especially for I/O operations. These programs have a low overhead because they not only avoid

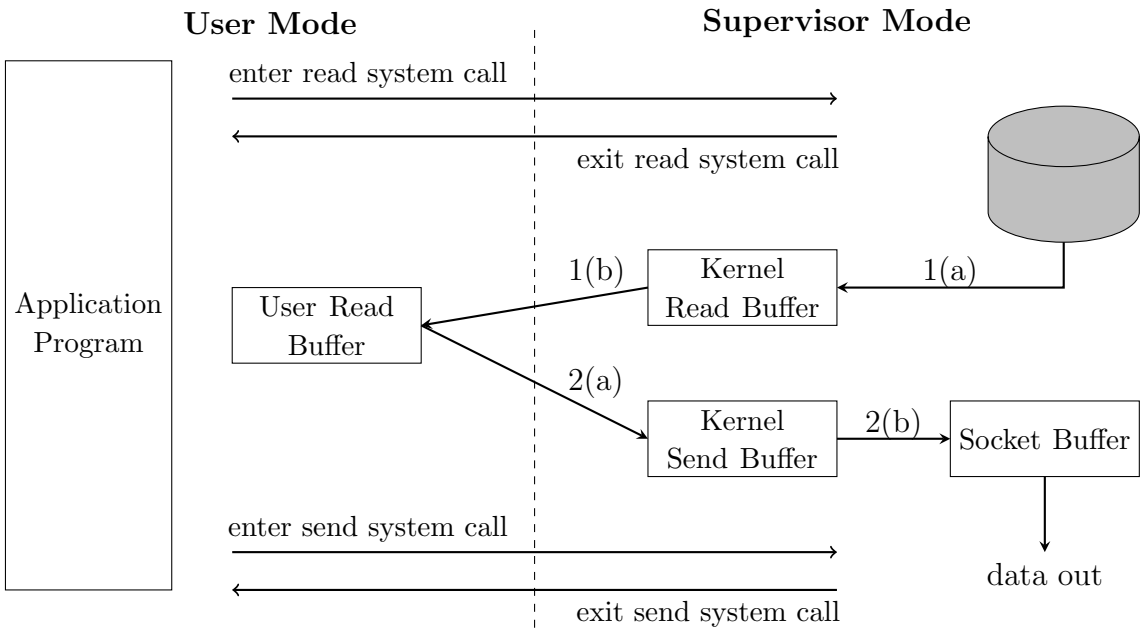

Figure 12.1: A Naïve Approach to Transferring Data: Data is being sent using two system calls. The buffers are depicted in the physical memory.

the creation of redundant data copies during information flow between user and kernel spaces, but also reduce the number of context switches between the user and supervisor modes. For example, consider a naïve method, illustrated in Figure 12.1, used by an application program to transfer data over the network.

1. A read system call is issued by the application program to copy data from the disk into a user buffer. This involves making two copy operations:

   (a) From the disk into a kernel buffer (e.g., by a DMA operation)

   (b) From the kernel buffer to the user buffer so that it is accessible to the application program

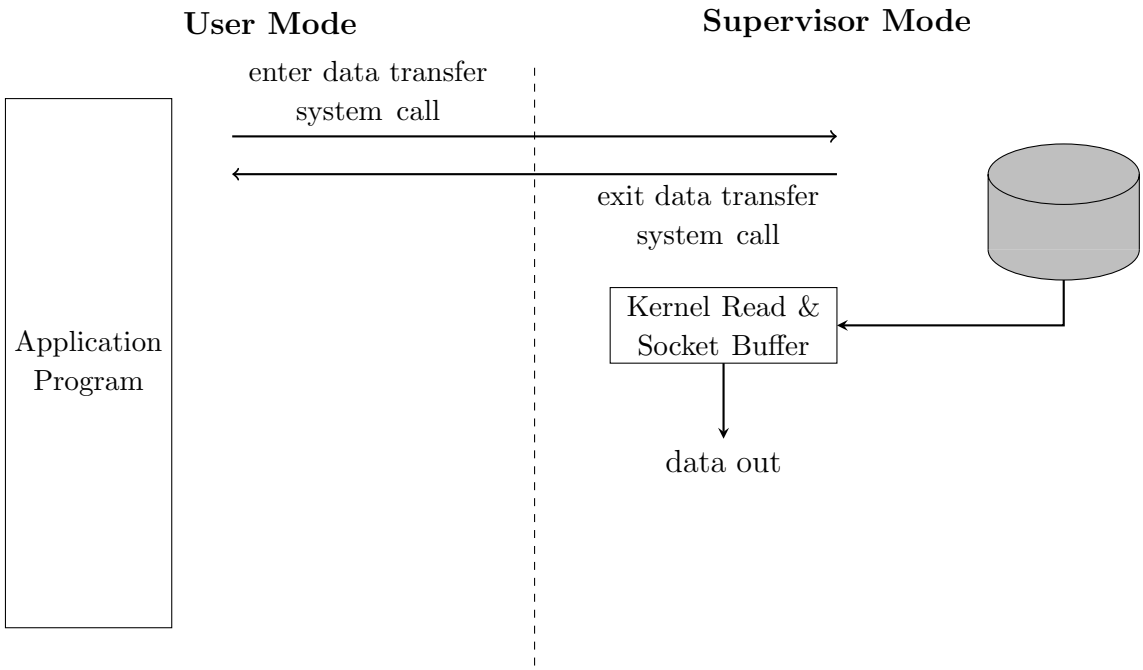

Figure 12.2: A Zero-Copy Approach to Transferring Data: Data is being sent using one system call. The buffer is depicted in the physical memory.

2. A send system call is issued by the application program to send the data to a network socket. This again involves making two copy operations:

   (a) From the user buffer (of the read system call) into a kernel buffer

   (b) From the kernel buffer to the socket buffer

There are four context switches involved here — each system call causes a switch from the user to supervisor mode at its beginning and back again at its end.

On the other hand, a zero-copy approach can enable data transfers using only one system call (i.e., two context switches) and one copy operation. Figure 12.2 illustrates this approach, where the kernel read buffer has been mapped to the same

physical location as the socket buffer using zero-copy. From the point of view of linear memory, these two buffers are distinct entities.

The functional correctness criteria for the zero-copy program are the same as that for a simple application-level copy program — at the end of the program's execution, the destination location in the linear address space must contain the same data as the source location in the linear address space. Also, the program must not modify the source data. Though somewhat similar in nature, the conditions under which this criteria are met for the zero-copy program are significantly more complex than those for a simple copy program. To put this complexity in perspective, we first describe the preconditions for correctness of an application-level copy program in Section 12.1. Then we describe our verification effort for the zero-copy program in Section 12.2. We conclude this chapter with a few remarks in Section 12.3.

12.1 Simple Copy Program

Consider the following naïve sub-routine `copyData` that copies `4n` bytes of data (`n` 32-bit integers) from a source linear address location `src` to a destination linear address location `dst`. Of course, `copyData` is an extremely simple sub-routine presented here for the purpose of illustration. One can imagine writing much efficient data-copying programs using `malloc`, for example.

```
void copyData (int* src, int* dst, unsigned int n) {
  int* dstEnd = dst + n;
  while (dst != dstEnd)
    *dst++ = *src++;
}
```

Compiling the above sub-routine using GCC (default optimization) results in an x86 machine-code sub-routine consisting of 15 instructions. The first source address is located in register `rdi`, the first destination address is in register `rsi`, and the number of doublewords (i.e., 32-bit values) to copy is in register `edx`.

```
     <copyData>:
 1:  55                      push   %rbp
 2:  48 89 e5                mov    %rsp,%rbp
 3:  85 d2                   test   %edx,%edx
 4:  74 1a                   je     <_copyData+0x22>
 5:  89 d0                   mov    %edx,%eax
 6:  48 c1 e0 02             shl    $0x2,%rax
 7:  66 90                   xchg   %ax,%ax
 8:  8b 0f                   mov    (%rdi),%ecx
 9:  48 83 c7 04             add    $0x4,%rdi
10:  89 0e                   mov    %ecx,(%rsi)
11:  48 83 c6 04             add    $0x4,%rsi
12:  48 83 c0 fc             add    $0xfffffffffffffffc,%rax
13:  75 ee                   jne    <_copyData+0x10>
14:  5d                      pop    %rbp
15:  c3                      retq
```

We verified this program using a similar approach to that followed for the word-count program, previously discussed in Section 11.2. We reasoned about the loop and the rest of the sub-routine separately, and used compositional verification to obtain the final theorem of correctness. Without describing the details of reasoning about `copyData`, we present this final theorem below, where `src-addr`, `dst-addr`, `prog-addr`, and `n` are the initial values in `rdi`, `rsi`, `rip`, and `edx` respectively, and `(clk n)` describes the number of instructions to be executed for the sub-routine to run to completion.

```
(defthm copyData-is-correct
  (implies
   (preconditions n src-addr dst-addr prog-addr x86)
   (and
    ;; Destination location after program's execution contains
    ;; the same data as the source location before program's
    ;; execution.
    (equal (destination n dst-addr (x86-run (clk n) x86))
           (source n src-addr x86))
    ;; Source location after program's execution contains the
    ;; same data as it did before program's execution.
    (equal (source n src-addr (x86-run (clk n) x86))
           (source n src-addr x86))
    ;; No model-related error was encountered during program's
    ;; execution.
    (equal (ms (x86-run (clk n) x86)) nil)
    (equal (fault (x86-run (clk n) x86)) nil))))
```

The conditions for the correctness of `copyData`, specified by the `preconditions` predicate, are simple and easily foreseeable. We divide these preconditions into labeled categories for convenience.

1. **Model-Related Preconditions:**

   - The initial x86 state satisfies its recognizer.
   - The initial x86 state is error-free (`ms` and `fault` fields contain the value `nil`).
   - The mode of operation of the x86 model is user-level mode (field `user-level-mode` contains the value `t`).

2. **Program:**

- The initial instruction pointer `rip` points to the first instruction of the sub-routine.
- The sub-routine is located at canonical linear addresses.

3. **Limit on Data to be Copied:**
   - The register `edx` contains a 32-bit unsigned integer.
4. **Source and Destination Linear Addresses:**
   - All the source and destination linear addresses are canonical.
5. **Stack:**
   - The stack linear addresses are canonical.
6. **Disjointness of Various Memory Regions:** The following linear memory locations are disjoint. These disjointness conditions prevent the write operations to the stack and the destination from over-writing the source data, the destination data, and the program.
   - The source, destination, and stack addresses are mutually disjoint.
   - The sub-routine's addresses are disjoint from the destination and the stack addresses.
7. **Return Address:**
   - The return address of the sub-routine is canonical.

A side note: precondition 3 is interesting — it places a limit of 2^{32} on the number of integers that can be copied. In practice, apart from obvious performance issues, such large amounts of data cannot be copied using this sub-routine on most systems. Why, then, is the formal analysis of `copyData` useful? Remember that this analysis has been done in isolation; assurances about systems as a whole can be obtained via compositional verification, if desired. Obtaining the preconditions for the correct operation of `copyData` assures us that the reason why large amounts of data cannot be copied using it in practice is not because of a bug in the sub-routine, but elsewhere — either in the calling program or because of resource limits imposed by the underlying system. Of course, this is obvious for the extremely simple `copyData`, but a similar argument can be made about the benefits of verifying (even in isolation) more complex and useful sub-routines.

12.2 Zero-Copy Program

Our zero-copy program operates under the following constraints:

1. The linear addresses of the first byte of data at the source and at the destination must be 1GB-aligned — their lower 30 bits must be zero.

2. The source and destination linear addresses constitute exactly one 1GB page each — that is, the linear addresses of the first byte of data at the source and at the destination should be the first addresses of two distinct 1GB linear memory regions in the linear address space. This mapping of 1GB of source and destination data is accomplished by two paging entries each — a `PML4TE`

that points to a PDPTE, which then maps a 1GB page.

3. The linear addresses of the paging entries, PML4TEs and PDPTEs, of the source and destination are mapped directly to physical addresses — i.e., the mapping of these paging entries from the linear address space to the physical address space is an identity function.

An implication of this configuration is that in order to map the destination's linear addresses to the source's physical addresses, the only entry that needs to be modified is the destination's PDPTE. See Figure 12.3 for an illustration.

Sub-routine rewire_dst_to_src, which implements the zero-copy program, is presented below; supporting sub-routines are presented in Appendix E. This sub-routine obtains the base address of the PML4 table from the control register cr3. It then walks the paging entries of the source linear address src_la by obtaining the address of the PML4TE (pml4e_src_pa) and then the PDPTE (pdpte_src_pa) to eventually obtain the corresponding physical address src_pa. Similarly, it walks the paging entries of the destination linear address dst_la by obtaining the addresses of the PML4TE (pml4e_dst_pa) and PDPTE (pdpte_dst_pa). It then copies the relevant field from the source's PDPTE located at address pdpte_src_pa to the destination's PDPTE at address pdpte_dst_pa using a helper sub-routine called copy_pdpte. After this modification of the destination PDPTE, the destination physical address modified_dst_pa is obtained; if this address is not equal to the source's physical address src_pa, then the sub-routine signals a failure by returning value 0, else value 1 is returned. At every step of the page walks, rewire_dst_to_src checks if any errors

are encountered (e.g., if the paging entries have their `P` (present) bit cleared) and returns value `-1` if so.

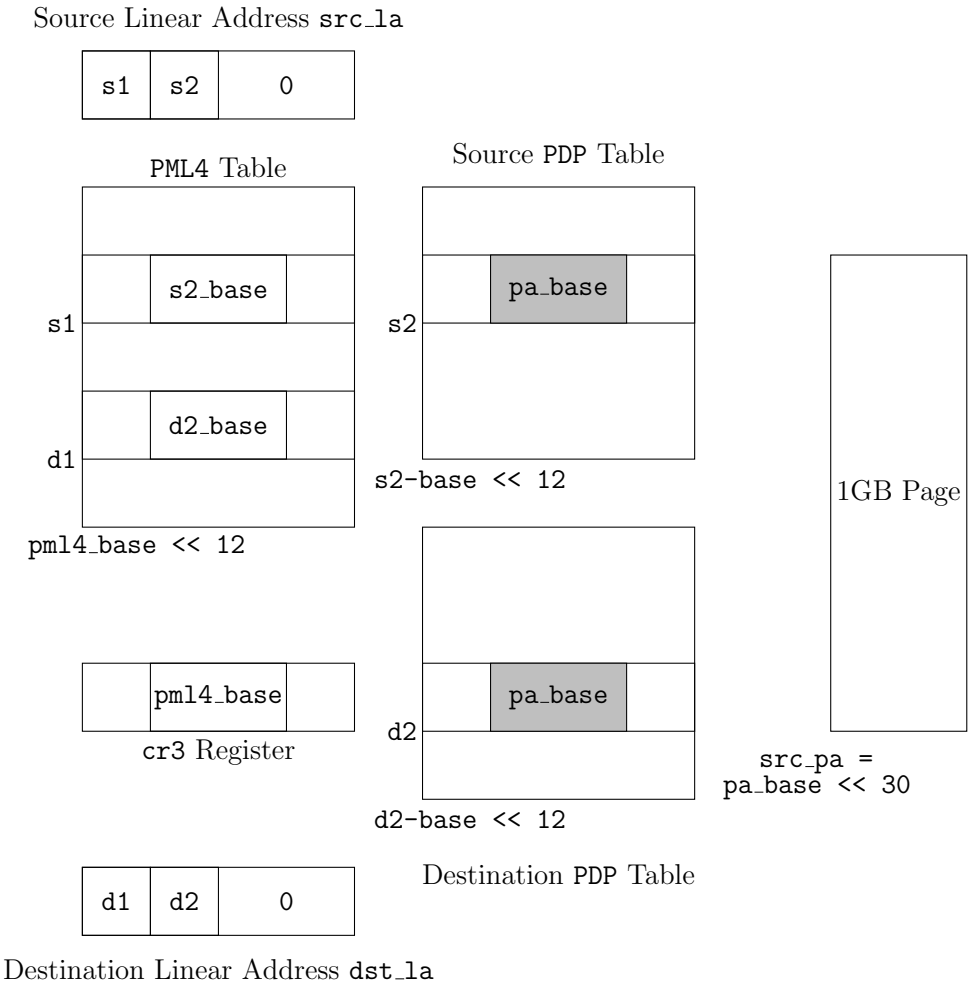

Figure 12.3: Destination `PDPTE` Modification in the Zero-Copy Program: The source and destination linear addresses `src_la` and `dst_la` are the first addresses of two distinct 1GB memory regions in the linear address space. The first two parts of `src_la` (`s1` and `s2`) and `dst_la` (`d1` and `d2`) are used as indices into the `PML4` table and `PDP` tables, respectively. The destination `PDPTE` points to the same 1GB page in the physical memory as the source `PDPTE` because it contains the same value, `pa-base`, in the appropriate field.

```
u64 rewire_dst_to_src (u64 src_la, u64 dst_la) {

  u64 cr3;
  u64 pml4e_src_pa, pdpte_src_pa, src_pa;
  u64 pml4e_dst_pa, pdpte_dst_pa, modified_dst_pa;
  u64 copy_pdpte_ret_stat;

  /* Get value in cr3 register. */
  __asm__ __volatile__
    ("mov %%cr3, %%rax\n\t"
     "mov %%rax, %0\n\t"
     : "=m"(cr3)
     :
     : "%rax");

  /* Obtain PDPTE address for src_la. */
  pml4e_src_pa   = pml4e_paddr(cr3, src_la);
  pdpte_src_pa   = pdpte_paddr(pml4e_src_pa, src_la);
  if (pdpte_src_pa == -1) return -1;
  src_pa         = paddr(pdpte_src_pa, src_la);
  if (src_pa       == -1) return -1;

  /* Obtain PDPTE address for dst_la. */
  pml4e_dst_pa   = pml4e_paddr(cr3, dst_la);
  pdpte_dst_pa   = pdpte_paddr(pml4e_dst_pa, dst_la);
  if (pdpte_dst_pa == -1) return -1;

  /* Map dst_la to src_pa. */
  copy_pdpte_ret_stat = copy_pdpte(pdpte_src_pa, pdpte_dst_pa);
  if (copy_pdpte_ret_stat == -1) return -1;

  /* Get physical address corresponding to dst_la. */
  modified_dst_pa = paddr(pdpte_dst_pa, dst_la);
  if (modified_dst_pa       == -1) return -1;

  if (modified_dst_pa == src_pa)
    return 1; /* Success */
```

```
  return 0;   /* Failure */
}
```

Compiling the zero-copy program (presented in its entirety in Appendix E) using GCC at the default optimization level results in the following machine-code program. Note that the helper sub-routines were inlined by the compiler.

```
    <rewire_dst_to_src>:
 1: 0f 20 d8                      mov %cr3,%rax
 2: 48 89 44 24 e8                mov %rax,-0x18(%rsp)
 3: 48 8b 54 24 e8                mov -0x18(%rsp),%rdx
 4: 48 89 f8                      mov %rdi,%rax
 5: 48 c1 e8 24                   shr $0x24,%rax
 6: 25 f8 0f 00 00                and $0xff8,%eax
 7: 48 81 e2 00 f0 ff ff          and $0xfffffffffffff000,%rdx
 8: 48 09 d0                      or %rdx,%rax
 9: 48 8b 00                      mov (%rax),%rax
10: a8 01                         test $0x1,%al
11: 0f 84 d2 00 00 00             je <rewire_dst_to_src+0x100>
12: 48 c1 e8 0c                   shr $0xc,%rax
13: 49 b8 ff ff ff ff ff 00 00 00 movabs $0xfffffffff,%r8
14: 48 89 f9                      mov %rdi,%rcx
15: 4c 21 c0                      and %r8,%rax
16: 48 c1 e9 1b                   shr $0x1b,%rcx
17: 81 e1 f8 0f 00 00             and $0xff8,%ecx
18: 48 c1 e0 0c                   shl $0xc,%rax
19: 48 09 c8                      or %rcx,%rax
20: 48 8b 00                      mov (%rax),%rax
21: 48 89 c1                      mov %rax,%rcx
22: 81 e1 81 00 00 00             and $0x81,%ecx
23: 48 81 f9 81 00 00 00          cmp $0x81,%rcx
24: 0f 85 94 00 00 00             jne <rewire_dst_to_src+0x100>
25: 48 89 f1                      mov %rsi,%rcx
26: 49 b9 00 00 00 c0 ff ff 0f 00 movabs $0xfffffc0000000,%r9
```

```
27: 48 c1 e9 24                     shr $0x24,%rcx
28: 49 21 c1                        and %rax,%r9
29: 81 e1 f8 0f 00 00               and $0xff8,%ecx
30: 48 09 d1                        or %rdx,%rcx
31: 48 8b 01                        mov (%rcx),%rax
32: a8 01                           test $0x1,%al
33: 74 70                           je <rewire_dst_to_src+0x100>
34: 48 c1 e8 0c                     shr $0xc,%rax
35: 48 89 f2                        mov %rsi,%rdx
36: 4c 21 c0                        and %r8,%rax
37: 48 c1 ea 1b                     shr $0x1b,%rdx
38: 81 e2 f8 0f 00 00               and $0xff8,%edx
39: 48 c1 e0 0c                     shl $0xc,%rax
40: 48 09 d0                        or %rdx,%rax
41: 48 ba ff ff ff 3f 00 00 f0 ff   movabs $0xfff000003fffffff,%rdx
42: 48 23 10                        and (%rax),%rdx
43: 4c 09 ca                        or %r9,%rdx
44: 48 89 10                        mov %rdx,(%rax)
45: 48 89 d0                        mov %rdx,%rax
46: 25 81 00 00 00                  and $0x81,%eax
47: 48 3d 81 00 00 00               cmp $0x81,%rax
48: 75 32                           jne <rewire_dst_to_src+0x100>
49: 48 b8 00 00 00 c0 ff ff 0f 00   movabs $0xfffffc0000000,%rax
50: 81 e6 ff ff ff 3f               and $0x3fffffff,%esi
51: 81 e7 ff ff ff 3f               and $0x3fffffff,%edi
52: 48 21 c2                        and %rax,%rdx
53: 4c 09 cf                        or %r9,%rdi
54: 31 c0                           xor %eax,%eax
55: 48 09 f2                        or %rsi,%rdx
56: 48 39 d7                        cmp %rdx,%rdi
57: 0f 94 c0                        sete %al
58: c3                              retq
59: 66 2e 0f 1f 84 00 00 00 00 00   nopw %cs:0x0(%rax,%rax,1)
60: 48 c7 c0 ff ff ff ff            mov $0xffffffffffffffff,%rax
61: c3                              retq
62: 0f 1f 84 00 00 00 00 00         nopl 0x0(%rax,%rax,1)
```

We now present the final theorems of correctness that we proved about this program in Section 12.2.1, and the preconditions for establishing these theorems in Section 12.2.2. In Section 12.2.3, we present an overview of our approach for reasoning about this program.

12.2.1 Properties Proved

Figure 12.4 provides a view of the linear memory after the zero-copy program runs to completion (that is, view of the memory in the final x86 state), assuming that the copy was done successfully. Given the preconditions in the previous section, we proved the following main properties about the zero-copy program.

1. The register `rax`, which contains the return value from `rewire_dst_to_src`, has `1` in the final x86 state. Recall that `rewire_dst_to_src` returns a value other than `1` — either `0` or `-1` — if an error is encountered over the course of execution of the program.

2. The 1GB of data at the destination's linear addresses in the final x86 state is the same as the 1GB of data at the source's linear addresses in the initial x86 state.

3. The 1GB of data at the source's linear addresses in the final x86 state is the same as the 1GB of data at the source's linear addresses in the initial x86 state.

4. The program in the final x86 state is the same as that in the initial x86 state.

5. The `ms` and `fault` fields are empty in the final x86 state.

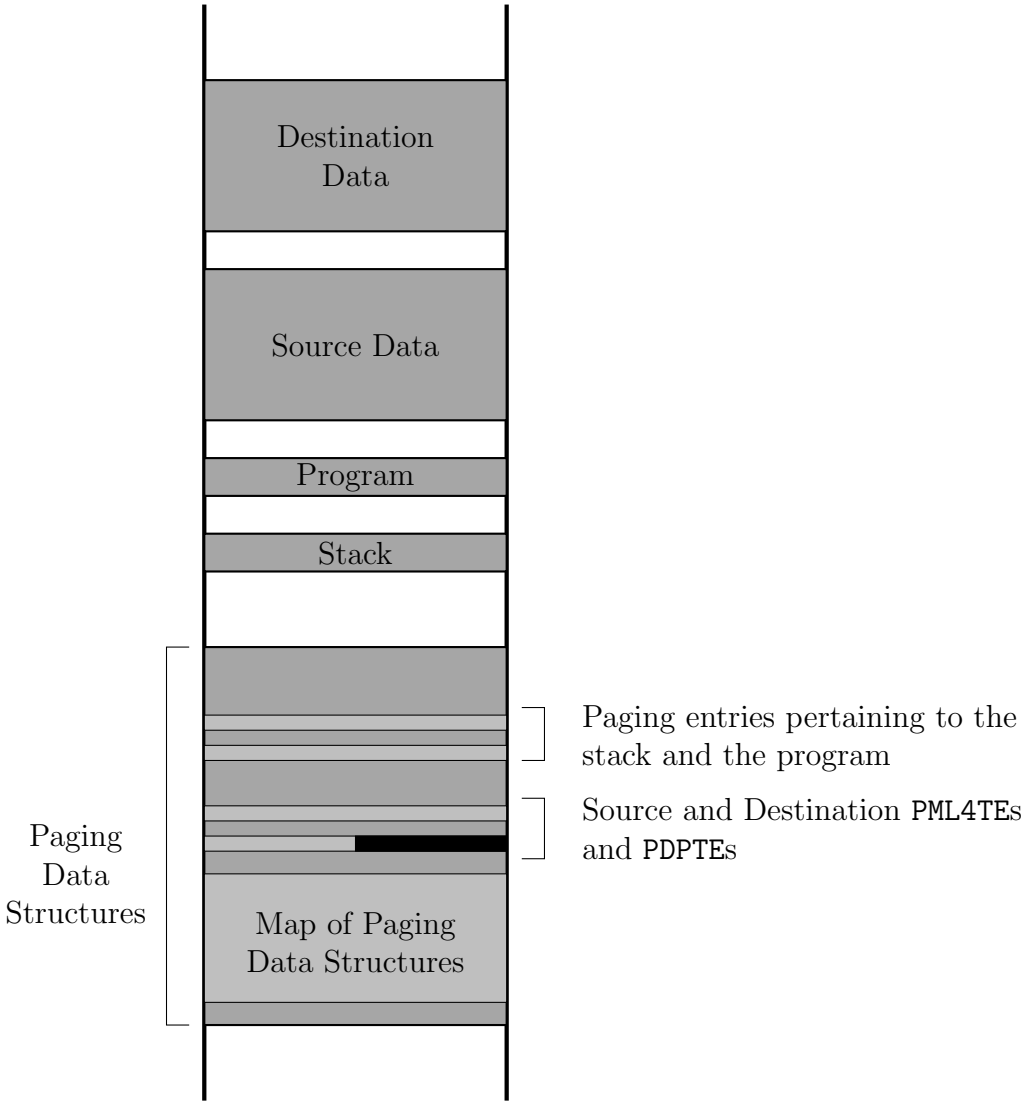

Figure 12.4: View of the Linear Memory after a Successful Run of Zero-Copy: A modification to the destination's `PDPTE` (shown in black) gives the illusion of destination data being separate from the source data in the linear address space. This diagram also illustrates the disjointness between the various structures in the memory, including the disjointness of entries within the paging data structures.

Note that this figure does not show updates made to the `A` and `D` flags during program execution, though they are accounted for in our formal analysis. Also, it is not drawn to scale, nor does it represent relative positions of various structures in the memory.

Properties 1 and 2 above denote that at the level of linear memory, the copy was done successfully. Properties 3 and 4 denote that changing the address mapping of the destination did not cause any side-effects in the contents of the source and the program, respectively. Property 5 denotes that no model-related error was encountered at any point during the program's execution.

12.2.2 Preconditions

The preconditions for the correctness of the zero-copy program are quite similar to those of the simple copy program of Section 12.1, though they are considerably more in number. This is essentially a consequence of the large number of disparate memory regions exposed to the supervisor-mode zero-copy program; these regions are generally inaccessible to application programs.

1. **Model-Related Preconditions:**

   - The initial x86 state satisfies its recognizer.
   - The initial x86 state is error-free (`ms` and `fault` fields contain the value `nil`).
   - The mode of operation of the x86 model is system-level marking mode (fields `user-level-mode` and `page-structure-marking-mode` have the values `nil` and `t` respectively). The current privilege level is the highest one (i.e., the two least-significant bits of the `CS` segment register are 0), which allows supervisor-mode operation.

2. **Program:**

- The initial instruction pointer `rip` points to the first instruction of the program.
- The program is located at canonical linear addresses.
- No errors are encountered while translating the program's linear addresses to physical addresses on behalf of instruction-fetch operations.

3. **Stack:**
   - The stack is located at canonical linear addresses.
   - No errors are encountered while translating the stack's linear addresses to physical addresses on behalf of both read and write operations.

4. **Source's Linear Addresses:**
   - The source is located at canonical linear addresses.
   - The linear address of the first byte of the source is 1GB-aligned.
   - No errors are encountered while translating the source's linear addresses to physical addresses on behalf of a read operation.

5. **Source's Paging Entries:**
   - The source's `PML4TE` is located at canonical linear addresses.
   - No errors are encountered while translating the linear addresses of the source's `PML4TE` to physical addresses on behalf of a read operation.
   - The field's of the source's `PML4TE` contain appropriate values — e.g., the `P` (present) bit of the entry is 1.

- The source's `PDPTE` is located at canonical linear addresses.
- No errors are encountered while translating the linear addresses of the source's `PDPTE` to physical addresses on behalf of a read operation.
- The field's of the source's `PDPTE` contain appropriate values — e.g., the `P` (present) bit of the entry is 1, the `PS` (page size) bit is 1, which indicates that this entry maps a 1GB page.

6. **Destination's Linear Addresses:**
   - The destination is located at canonical linear addresses.
   - The linear address of the first byte of the destination is 1GB-aligned.
   - No errors are encountered while translating the destination's linear addresses to physical addresses on behalf of a read operation.

7. **Destination's Paging Entries:**
   - The destination's `PML4TE` is located at canonical linear addresses.
   - No errors are encountered while translating the linear addresses of the destination's `PML4TE` to physical addresses on behalf of a read operation.
   - The field's of the destination's `PML4TE` contain appropriate values — e.g., the `P` (present) bit of the entry is 1.
   - The destination's `PDPTE` is located at canonical linear addresses.
   - No errors are encountered while translating the linear addresses of the destination's `PDPTE` to physical addresses on behalf of both a read and a write operations.

- The field's of the destination's `PML4TE` contain appropriate values — e.g., the `P` (present) bit of the entry is 1, the `PS` (page size) bit is 1, which indicates that this entry maps a 1GB page.

8. **Address Mapping of Paging Entries:**

   - The linear addresses of the source and destination `PML4TE`s and `PDPTE`s are mapped directly to physical addresses — i.e., the mapping is an identity function.

   - The source and destination `PML4TE`s and `PDPTE`s are all distinct. This condition is less restrictive than it may appear at first glance because it is about the individual entries rather than the paging structures. For example, it does not preclude two `PDPTE`s from belonging to the same `PDP` table, nor does it preclude paging structure configurations where `PML4` and `PDP` tables overlap with other paging structures, and so on.

9. **Disjointness of Various Memory Regions:**

   - The physical addresses of the source and destination `PML4TE`s and `PDPTE`s are disjoint from the physical addresses of their own and each other's translation-governing entries.

   - The physical addresses of the translation-governing entries of the stack and program are disjoint from the physical addresses of the source and destination `PML4TE`s and `PDPTE`s.

- The physical addresses of the stack, the program, the source, and the paging data structures are mutually disjoint.

10. **Return Address for `rewire_dst_to_src`:**

    - The return address is a canonical linear address.

11. **Miscellaneous:**

    - MBZ ("Must Be Zero") reserved portions (e.g., the higher bits of the `cr3` control register) are zero.

We ensure that these preconditions are reasonable (i.e., not restrictive or contradictory) by performing concrete tests, as described previously in Section 9.3. These preconditions evaluated to `t` for different witness states; in addition to different locations of the program, stack, and data, we used different configurations of the paging structures in the memory to create these different witnesses.

The discovery of these preconditions would have been more challenging had we not performed our analysis in the system-level non-marking mode of operation first. Then, we ported our proofs to over to the system-level marking mode, the true specification of the x86 ISA, relatively easily. Details about this process are presented in the following section.

12.2.3 Proof Approach

In this section, we present an outline of our zero-copy verification effort, without going into specifics about how the program's symbolic simulation was controlled

using our general-purpose lemma libraries. The reader is encouraged to refer back to Chapter 10 for such details.

Our zero-copy program is structurally simple. It does not contain any loops, though it does contain many branches on account of error checking. Since we are interested in reasoning about executions of this program that lead to a successful copy, we specify preconditions that allow only those branches to be taken that indicate the absence of an error. Thus, the control flow of this program is predictable during reasoning. Arriving at the final theorems of correctness, presented in the previous section, is simply a matter of symbolically simulating the program till completion, that is, when it reaches the final state after executing the last instruction. We then project out relevant components of this final x86 state — such as the `rax` register for Property 1, linear addresses pointing to 1GB of destination and source data for Properties 2 and 3, linear addresses pointing to the program for Property 4, and the `ms` and `fault` fields for Property 5 — in order to assert that the correctness conditions hold.

All this being said, zero-copy has supervisor privileges and access to a large x86 system state, which makes the discovery of preconditions challenging. This program owes its complexity to an update it makes to a paging entry — though this in itself is nothing more complicated than a data structure write, it has the effect of changing the linear memory abstraction *during* the program's execution.

We first reasoned about the zero-copy program in the system-level non-marking mode and then ported the proof over to the system-level marking mode of operation. Recall that these two modes are exactly the same, except that the updates to

A and D flags during page walks are suppressed in the non-marking mode. Reasoning about a program in the non-marking mode allows a user to focus on the ramifications of *explicit* updates made to the paging structures by a program, without the side-effect updates to A and D flags muddling up the verification process. Porting over a proof done in the non-marking mode to the marking mode is straightforward — one simply needs to discover and add the extra preconditions required to account for these side-effect updates. Thus, formal analysis of a program in the non-marking mode prior to that in the marking mode simplifies the discovery of preconditions and the verification process in general.

Reasoning in the Non-Marking Mode

The symbolic simulation of `rewire_dst_to_src` in the non-marking mode is straightforward, similar to that of an application program. Our libraries provide rules to "unwind" our x86 ISA interpreter (for example, the step function opener rule, previously discussed in Section 9.1), which, along with Read-over-Write, Write-over-Write, Writing-the-Read, and State Well-Formedness rules, are used to obtain a symbolic x86 state that captures the behavior of this program.

Note that the modification of the destination's PDPTE made by this program does not change the *location* of the paging structures in the memory, merely the *contents* of this one entry (refer back to Figure 12.3 for an illustration). This is because this PDPTE maps a page instead of referring to another paging entry — had the latter been true, modifying the PDPTE would have changed the set of addresses where the paging structures were located. Consequently, we can establish the following

properties:

1. Given the precondition that the paging structures were disjoint from the other memory regions (such as the program, stack, source, and destination locations) in the initial x86 state, this disjointness also holds over the course of execution of this program, despite the modification to the destination's `PDPTE`.

2. Given the precondition that the destination's `PDPTE` is distinct from the translation-governing paging entries of the source locations, the program, and the stack in the initial x86 state, the modification of the contents of the destination's `PDPTE` does not affect the address mapping of these memory regions over the course of execution of this program.

3. The destination's `PDPTE` in the final x86 state points to the same page as the source's `PDPTE` in the final x86 state.

The first two properties above help in inferring Properties 3 and 4 of this program — that is, the source data and the program in the final x86 state are the same as that in the initial x86 state. The last property above implies Property 2 — the linear addresses corresponding to the destination data map to the physical addresses of the source data. For establishing the Properties 1 and 5 of this program, we simply project out the `rax`, `ms`, and `fault` from the final x86 state, and prove that they are equal to 1, `nil`, and `nil`, respectively.

Reasoning in the Marking Mode

Conceptually, we divide `rewire_dst_to_src` into two parts in this mode of operation. The first part (consisting of the first 42 instructions) does not explicitly modify the paging entries but simply accesses them, both for address translations on behalf of instruction fetches as well as for reading the source and destination paging entries. The second part begins at an instruction in `copy_pdpte` (specifically, the 43^{rd} instruction) that copies the appropriate field of the source's `PDPTE` to the destination's `PDPTE`.

We employ our libraries that provide conditional congruence-based rewriting, previously discussed in Section 10.2.2. Recall that the purpose of using conditional congruence-based rewriting is to facilitate reasoning in the marking mode in an analogous manner to that in the non-marking mode *if* the updates to the `A` and `D` flags do not impact the behavior of the program — if that is indeed the case, then these updates are ignored in the marking mode of operation. This has the benefit of not needing to establish that these side-effect updates did not modify the locations or contents of memory regions of interest for every memory read operation, thereby speeding up symbolic simulation.

All the instructions in the first part preserve the equivalence relation `xlate-equiv-memory` because the only changes to the paging structures in this part are to the `A` and `D` flags. Functions that denote instruction fetches — `rb` — and the location of the program — `program-at` — are *automatically* rewritten to `rb-alt` and `program-at-alt` respectively, given the disjointness of the program and the paging structures. Thus, every instruction fetch is done automatically, without having to

prove that the modifications made to the memory by the previous instruction fetch did not interfere with the fetch of the current instruction. In this way, the symbolic simulation of the first part proceeds in a similar manner to that in the non-marking mode.

In the second part, the linear memory abstraction is changed due to the write to the destination's `PDPTE`. The x86 state obtained after the symbolic simulation of the first part and the x86 state obtained after the first instruction of the second part do not satisfy `xlate-equiv-memory` — the destination's `PDPTE`s in the old and new paging structures are not equal, even modulo the `A` and `D` flags. However, the disjointness of the program with the paging structures is still preserved in the second part for the same reason as before in the non-marking mode. This again facilitates automatic rewriting of `rb` and `program-at` to `rb-alt` and `program-at-alt` respectively, which allows symbolic simulation to proceed as usual.

Similar to the non-marking mode, once we obtain the final x86 state that captures the behavior of this program, we prove this program's final theorems of correctness by using our non-interference rules and by projecting out the relevant components of this state.

12.3 Remarks

In addition to the properties listed in Section 12.2.1, we could also have proved that the execution of this program leaves the entire physical memory, except for the paging structures, unchanged, and that the only change at the level of linear memory, apart from the paging structures, is at the destination addresses. Another interesting

property would be that the only modifications to the paging structures made by this program (not counting the side-effect updates to `A` and `D` flags) are to the `PDPTE` of the destination. We get some confidence that this last property is true — though we have not proved it formally — by observing the symbolic expression that describes the behavior of zero-copy in terms of updates made to the initial x86 state. This symbolic expression is obtained over the course of reasoning about zero-copy, and it contains only two writes to the linear memory (again, not counting the side-effect updates to `A` and `D` flags) — the first one to the stack (see the second x86 instruction in the zero-copy machine-code program that stores the value of the `rax` register to a location on the stack) and the second one to the destination `PDPTE`. However, at this time, we focused only on proving that the program and the source data were unmodified after the program ran to completion.

We focused on non-overlapping source and destination locations in this case study, but memory-copy routines like `memmove` allow copying from overlapping source and destinations. Precondition discovery for the correctness of such programs would be a little more challenging than that for our zero-copy program.

It remains to be seen how the verification of the zero-copy program — and more generally, other supervisor-mode programs — will proceed when our x86 ISA model is extended to support features such as caches, TLBs, interrupts, etc. This case study serves as a baseline for those future efforts in that it states precisely the properties that programmers expect to be true, irrespective of these other ISA features. This knowledge will help in discovering invariants involving these features that must hold for the program to be correct.

Our zero-copy program modifies the paging structures, but it does not change their location in the memory, merely their content. This makes the proof of key disjointness properties relatively straightforward, thereby simplifying symbolic simulation. However, our general-purpose libraries provide lemmas that allow reasoning about overlap or disjointness of memory regions in general, and they can be used to reason about supervisor-mode programs that change the location of the paging structures as well (e.g., by changing a paging entry to refer to another paging structure). Our libraries also include lemmas that describe the x86 state after a page walk precisely, and they can be used to analyze programs that read or directly modify the `A` and `D` flags of paging entries.

Even though zero-copy is simple by the standards of a kernel program, it is representative of supervisor-mode programs. This case study demonstrates that our reasoning framework is capable of mechanically reasoning about system programs, even when they directly interact with the paging structures.

Part V

Epilogue

Chapter 13

Conclusions and Future Work

This dissertation has provided evidence that formal verification of programs can be done at the level of machine code, without making any simplifications in the specification of the instruction-set architecture — even when the ISA is as complicated as that of the x86 processors. Our accurate formal x86 ISA model compels the user to consider all the ISA features during program analysis, and not merely those perceived to be "interesting". The completeness of the formal model can add complexity to the program verification process, and in this dissertation, we have demonstrated how our general-purpose lemma libraries can make mechanical verification of machine-code programs tractable.

Our x86 machine-code analysis framework incorporates the design goals of *accuracy*, *execution and reasoning efficiency*, and *usability* that were laid down at the beginning of this research. We ensure the accuracy of our x86 ISA specification by running co-simulations to validate it against a real x86 processor. For efficiency, we use abstraction techniques so that a naïve definition can be used for reasoning and an optimized but equivalent one can be used for execution. As a part of the usability design goal, our x86 model offers different modes of operation to allow the user to choose the depth of analysis desired. The analysis framework is documented both

to aid future developers as well as users. The documentation is easily accessible online [50] as part of the ACL2+Community books documentation. Also, all the libraries developed over the course of this dissertation are available under a permissive 3-Clause BSD license [51].

We summarize the main contributions of this dissertation below.

Formal, Executable Specification of the x86 ISA This project has yielded a formal, executable model of the x86 ISA (IA-32e mode). The x86 model contains a specification of 413 instruction opcodes, which include arithmetic, control-flow, floating-point, and some system-mode instructions. It captures the entire IA-32e execution environment, including system state.

- The x86 model is an accurate reference of the x86 ISA. Intel's Software Developer's Manuals were used as the main reference, and as far as possible, the specification functions in our x86 model point to the relevant sections of these manuals for transparency and to facilitate code reviews. In order to gain more trust in the accuracy of our model, we perform model validation via co-simulations.
- The x86 model is equipped with various tools that enable its use as an instruction-set simulator. We provide a program loader and dynamic instrumentation libraries to inspect the behavior of a program during simulations with concrete data.

Reasoning Framework for x86 Machine-Code Analysis We provide general lemma

libraries in each mode of operation of the x86 model. These libraries aid in automating symbolic simulation of x86 machine-code programs.

- The libraries in the user-level mode include lemmas to aid in reasoning about non-determinism inherent in the interactions of an application program with an external environment.
- In the system-level mode, we provide a library to reason about paging data structures.
- These lemmas are true for every program under consideration. Additionally, they describe the properties of the x86 ISA.

Verification of Application and System Programs Using our analysis framework, we have formally analyzed both application and system programs, and presented the following three as case studies in this dissertation.

- Verification of the *pop-count* application program shows that we can use a bit-blasting tool to reason about straight-line x86 machine-code programs completely automatically.
- Verification of the *word-count* program shows that our framework is capable of reasoning about application programs that make system calls.
- Verification of the *zero-copy* program demonstrates our framework's capabilities to reason about accesses and updates to low-level system data structures and how these updates impact the view of the processor available to programs.

13.1 Future Work

There are many opportunities for future research based on the work presented in this dissertation. A direct application of this work can be to extend (if necessary) and employ our x86 machine-code verification framework to formally analyze various kinds of software — for instance, self-modifying programs, linkers and loaders, compilers, etc. Below, we discuss just a few possible directions where our research can be useful — our focus is on long-term projects here.

Operating System Verification The x86 ISA model can be extended to support booting a mainstream OS. This task will not only increase confidence in the accuracy of our model, but it will also allow early detection of OS issues, such as dependencies on non-portable or undefined behaviors that might have security-related implications. Eventually, this will facilitate reasoning about kernel programs that are used routinely. Note that supporting a mainstream OS will be a formidable long-term project, and it will involve specifying complicated ISA features, such as more supervisor-mode instructions, I/O capabilities, interrupts and exceptions, TLBs and caches, etc. Additionally, one would need to investigate how to adapt our verification strategies to take all these new features into account during program verification.

Another related direction of research can be to prove that given an OS, the user-level mode of operation of our x86 model is a valid abstraction of the system-level mode (i.e., the "real" x86 ISA specification). This would likely involve many statements about various components of the OS and the ISA. An example is a statement about the system call service — the OS implementation should provide the same functionality as that described by the extended instruction semantic function

of the `syscall` instruction in the user-level mode. Another example is that the OS should ensure that an application program is being run in its own dedicated address space, i.e., the entries in the memory-management data structures pertaining to the application program are well-formed in some defined manner. Even formulating such properties in a mathematical logic will be extremely beneficial because they will state precisely the notion of correctness that programmers expect from an operating system.

Reasoning about the Memory System Our x86 model can be extended to include the entire memory hierarchy (e.g., internal caches and buffers). Doing so will give a complete specification of how memory reads and writes are resolved by the processor. We can then work at obtaining various kinds of assurances provided by the memory hierarchy. For example, are the caches and buffers (mostly) transparent to the programs? This would also enable reasoning about programs that use cache-management instructions, such as `INVD`, `PREFETCHh`, `CLFLUSH`, etc.

Multi-process/threaded Program Verification Our model specifies a uniprocessor x86, but it can be extended to specify multiple processors in order to reason about multi-process/threaded program behavior. This undertaking will result in a precise model of the x86 memory ordering mechanism [148, 150] in the executable mathematical logic of ACL2, thereby facilitating investigations about concurrency-related issues — system memory coherency and cache consistency — via both concrete and symbolic execution methods. This project would also require investigations into adapting our verification strategies to account for multi-process/threaded program behavior.

Micro-architecture Verification The external interface of a micro-architecture implementation is defined by the ISA; for example, one or more micro-operations implement an ISA-level instruction. Thus, our x86 ISA specification can serve as a build-to specification for micro-architecture verification, as stated in a paper [87] about micro-code verification by Centaur Technology [24].

Firmware Verification Firmware is low-level software that controls hardware directly; thus, it is hardware-specific software. Firmware is increasingly being used to implement operations previously provided by hardware. Firmware attacks [63, 100] can compromise the security of the entire system, especially since firmware forms an integral part of the Trusted Computing Base (TCB). Thus, formal analysis of firmware is of paramount importance, and it has received some attention in the community. For example, Horn et al. have used model checking for firmware validation [101]. However, their work requires both hardware and firmware to be described in the same language, either C or SystemC. Our x86 ISA model and supporting libraries are an excellent fit for this problem because they allow analysis to be performed at the level of the ISA, which is suitable for describing both hardware and software interfaces.

User-friendly Program Analysis Though our work involves automating some x86 machine-code proofs, a user still requires some knowledge of the x86 architecture. It is desirable to make machine-code verification accessible to users unfamiliar with the specifics of the processor. To this end, a future project can apply and/or extend existing decompilation tools and techniques [3, 125] to work with our x86 ISA model for program verification. Also, discovering preconditions for a program's correctness

is one of the most challenging parts of program analysis. We use ACL2 features like `break-rewrite` to manually discover preconditions, as described in Chapter 9. We can automate this process by writing ACL2 tools that use the output from `break-rewrite` to make suggestions to the user. Use of our framework by others in the community will suggest more ways to make program analysis user-friendly. For example, Kestrel Technology [42] is already using our framework for a project involving analysis of x86 binaries[1].

The availability of our machine-code analysis framework as a part of the ACL2 Community books, along with its documentation, will directly enable its use for future research projects. Thus, in addition to its direct contributions, this work paves the way for research opportunities that would otherwise have been difficult to pursue.

13.2 Concluding Remarks

Our research extends the state-of-the-art in software analysis by providing a means to mechanically prove or disprove arbitrarily complex properties of x86 programs without compromising on precision. We have developed a formal verification framework for x86 machine code that meets the design goals that practical, general-purpose analysis tools must strive to achieve. This framework is suitable for use in a large variety of future projects involving both software and hardware verification. Considering that the work presented in this dissertation — including case studies

[1]Personal communication with Eric Smith, Senior Computer Scientist at Kestrel

that illustrate the use of our framework for program analysis — was the result of one dissertation over the course of a handful of years, perhaps it is fair to say that the notion of thorough formal analysis of computing systems being impractical is dispelled.

Appendices

Appendix A

Representation of x86 State

Below, we present the ACL2 events that introduce the x86 state. See Chapter 5 of this dissertation to put these events in context.

A.1 x86 Concrete State

The `defstobj` form that introduces that x86 concrete stobj is presented below; details like renaming accessor and updater functions, are elided from the event below to save space.

```
(defstobj x86$c
  ;; General-purpose registers
  (rgf$c :type (array (signed-byte 64)
                      (#.*64-bit-general-purpose-registers-len*))
         :initially 0 :resizable nil)
  ;; Instruction pointer
  (rip$c :type (signed-byte 48) :initially 0)
  ;; Rflags register is defined as a 32-bit field instead of as a
  ;; 64-bit field in order to avoid bignum creation.  The top
  ;;  32 bits are reserved and writing to them is an error.
  ;; Similar to the optimization for the RIP, we check for the
  ;; well-formedness of a value before writing it to this register.
  (rflags$c :type (unsigned-byte 32)
            ;; Bit 1 is always 1.
            :initially 2)
  ;; User Segment Registers
```

```
(seg-visible$c :type (array (unsigned-byte 16)
                            (#.*segment-register-names-len*))
               :initially 0 :resizable nil)
(seg-hidden$c :type (array (unsigned-byte 112)
                           (#.*segment-register-names-len*))
              :initially 0 :resizable nil)
;; System Table Registers (GDTR and IDTR)
(str$c :type (array (unsigned-byte 80)
                    (#.*gdtr-idtr-names-len*))
       :initially 0 :resizable nil)
;; System Segment Registers (Task Register and LDTR)
(ssr-visible$c :type (array (unsigned-byte 16)
                            (#.*ldtr-tr-names-len*))
               :initially 0 :resizable nil)
(ssr-hidden$c :type (array (unsigned-byte 112)
                           (#.*ldtr-tr-names-len*))
              :initially 0 :resizable nil)
;; Control registers
(ctr$c  :type (array (unsigned-byte 64)
                     (#.*control-register-names-len*))
        :initially 0 :resizable nil)
;; Debug registers
(dbg$c :type (array (unsigned-byte 64)
                    (#.*debug-register-names-len*))
       :initially 0 :resizable nil)
;; FPU 80-bit data registers: the MMX registers (MM0 through MM7)
;; are aliased to the low 64-bits of the FPU data registers.
(fp-data$c :type (array (unsigned-byte 80)
                        (#.*fp-data-register-names-len*))
           :initially 0 :resizable nil)
;; FPU 16-bit control register
(fp-ctrl$c :type (unsigned-byte 16) :initially 0)
;; FPU 16-bit status register
(fp-status$c :type (unsigned-byte 16) :initially 0)
;; FPU 16-bit tag register
(fp-tag$c :type (unsigned-byte 16) :initially 0)
;; FPU 48-bit last instruction pointer
```

```
(fp-last-inst$c :type (unsigned-byte 48) :initially 0)
;; FPU 48-bit last data (operand) pointer
(fp-last-data$c :type (unsigned-byte 48) :initially 0)
;; FPU 11-bit opcode
(fp-opcode$c :type (unsigned-byte 11) :initially 0)
;; XMM 128-bit data registers
(xmm$c :type (array (unsigned-byte 128)
                    (#.*xmm-register-names-len*))
       :initially 0 :resizable nil)
;; MXCSR
(mxcsr$c :type (unsigned-byte 32)
         ;; Bits 7 through 12 are the individual masks for the
         ;; SIMD floating point exceptions.  These are set upon
         ;; a power-up or reset.
         :initially 8064)
;; Model-specific registers
(msr$c :type (array (unsigned-byte 64)
                    (#.*model-specific-register-names-len*))
       :initially 0
       :resizable nil)
;; Time- and Space-Efficient Memory Model
(mem-table$c :type (array (unsigned-byte #.*mem-table-size-bits+1*)
                          (#.*mem-table-size*))
             :initially 1
             :resizable nil)
(mem-array$c :type (array (unsigned-byte 8)
                          (#.*initial-mem-array-length*))
             :initially 0
             :resizable t)
(mem-array-next-addr$c :type (integer 0 #.*mem-table-size*)
                       :initially 0)

;; Other fields that are an artifact of our model rather than that
;; of the x86 ISA:
(ms$c :type t :initially nil)
(fault$c :type t :initially nil)
(env$c :type (satisfies env-alistp) :initially nil)
```

```
  (undef$c :type t :initially 0)
  (user-level-mode$c
   :type (satisfies booleanp) :initially t)
  (page-structure-marking-mode$c
   :type (satisfies booleanp) :initially t)
  (os-info$c
   :type (satisfies keywordp) :initially :linux))
```

Admitting this concrete stobj also introduces the stobj creator and recognizer, and each field's accessors, updaters, and recognizers. For example, the accessor for the field representing the general-purpose registers is `rgf$ci` and the updater is `!rgf$ci`. The native recognizer for the stobj is called `x86$cp-pre` and the native creator is called `create-x86$c`. The recognizer for a well-formed concrete x86 state is defined by `x86$cp`, which is a conjunction of `good-memp` (the predicate that recognizes that the memory model is correct) and `x86$cp-pre`.

A.2 x86 Abstract State

In this section, we present some key events related to introducing the x86 abstract stobj. The definition of the abstract stobj corresponding to the concrete stobj `x86$c` is as follows.

```
(defabsstobj x86
  ;; The concrete stobj corresponding to x86 is x86$c.
  :concrete x86$c
  ;; The recognizer for the abstract stobj is x86p, which is
  ;; x86$ap in the logic and x86$cp-pre (the native concrete
  ;; stobj recognizer) for execution.
  :recognizer (x86p :logic x86$ap :exec x86$cp-pre)
  ;; The initial stobj (with default values) is defined by
```

```
;; create-x86.
:creator (create-x86 :logic create-x86$a :exec create-x86$c)
;; The correspondence between x86 and x86$c is defined by
;; the corr function.
:corr-fn corr
:exports
;; The exports define the accessors and updaters for the abstract
;; stobj.  For example, rgfi* is the top-level accessor for the
;; general-purpose registers --- it is defined as rgf$ai in logic
;; and rgf$ci (the concrete stobj's accessor for the
;; general-purpose registers) for execution.
((rgfi* :logic rgf$ai :exec rgf$ci)
 (!rgfi* :logic !rgf$ai :exec !rgf$ci)
 (memi* :logic mem$ai :exec mem$ci)
 (!memi* :logic !mem$ai :exec !mem$ci :protect t)

 ;; Exports for other fields elided...

 (os-info* :logic os-info$a :exec os-info$c)
 (!os-info* :logic !os-info$a :exec !os-info$c)))
```

Recall that memory and other array fields of the concrete stobj are modeled as *records* in the abstract stobj to enable efficient reasoning. The abstract accessor functions are defined in terms of `g`, the record accessor function. Similarly, the abstract updater functions are defined in terms of `s`, the record updater function.

```
(defun mem$ai (i x86)
  (g i (nth *memi* x86)))

(defun !mem$ai (i v x86)
  (update-nth *memi* (s i v (nth *memi* x86)) x86))
```

The correspondence between the concrete and abstract x86 states is specified by the function `corr`, which is defined as follows.

```
(defun corr (c a)
  (and
   ;; "c" satisfies the recognizer for a well-formed concrete x86
   ;; state (which includes the well-formedness of the memory).
   (x86$cp c)
   ;; "a" satisifes the recognizer for a well-formed abstract x86
   ;; state.
   (x86$ap a)
   ;; Each field in the concrete x86 state corresponds to a
   ;; component in the abstract x86 state.
   (corr-rgf c (nth *rgfi* a))
   (corr-mem c (nth *memi* a))

    ;; Correspondence functions for other fields elided...

   (equal (nth *os-info$c* c)
          (nth *os-info* a))))
```

The function `corr-mem` specifies the correspondence of memory in the concrete and abstract stobjs — it says that looking up a valid address of the memory in the concrete stobj returns the same value as looking it up in the abstract stobj. The correspondence predicate for other fields is defined analogously.

```
(defun-sk corr-mem (x86$c field)
    (forall i
            (implies (and (natp i)
                          (< i *mem-size-in-bytes*))
                     (equal (mem$ci i x86$c)
                            (g i field)))))
```

As mentioned in Section 5.2.2, we need three kinds of theorems to admit an abstract stobj — correspondence, preservation, and guard theorems. We present examples of these theorems for the memory updater functions.

```
(defthm !memi*{correspondence}
  (implies (and (corr x86$c x86)
                (x86$ap x86)
                (natp i)
                (< i *2^52*)
                (unsigned-byte-p 8 v))
           (corr (!mem$ci i v x86$c)
                 (!mem$ai i v x86)))
  :rule-classes nil)

(defthm !memi*{preserved}
  (implies (and (x86$ap x86)
                (natp i)
                (< i *2^52*)
                (unsigned-byte-p 8 v))
           (x86$ap (!mem$ai i v x86)))
  :rule-classes nil)

(defthm !memi*{guard-thm}
  (implies (and (corr x86$c x86)
                (x86$ap x86)
                (natp i)
                (< i *2^52*)
                (unsigned-byte-p 8 v))
           (and (unsigned-byte-p 52 i)
                (unsigned-byte-p 8 v)
                (n08p v)
                (x86$cp x86$c)))
  :rule-classes nil)
```

Appendix B

x86 Instruction Semantic Function

We present the instruction semantic function for the following opcodes to give an idea of the complexity of the specification.

```
Opcodes 00, 01: ADD
Opcodes 08, 09: OR
Opcodes 10, 11: ADC
Opcodes 18, 19: SBB
Opcodes 20, 21: AND
Opcodes 28, 29: SUB
Opcodes 30, 31: XOR
Opcodes 38, 39: CMP
Opcodes 84, 85: TEST
```

We have omitted definitions of supporting functions here, and the reader is not expected to gain a precise understanding of the behavior of these instructions. A reader interested to know more about this or other instructions is referred to our source code, available online [51].

```
(defun x86-add/adc/sub/sbb/or/and/xor/cmp/test-e-g
  (operation start-rip temp-rip prefixes
             rex-byte opcode modr/m sib x86)
  ;; Guards elided.
  (b* ((ctx 'x86-add/adc/sub/sbb/or/and/xor/cmp/test-E-G)
       (r/m (the (unsigned-byte 3) (mrm-r/m modr/m)))
       (mod (the (unsigned-byte 2) (mrm-mod  modr/m)))
```

```
(reg (the (unsigned-byte 3) (mrm-reg  modr/m)))
(lock? (eql #.*lock*
            (prefixes-slice :group-1-prefix prefixes)))
((when (and lock? (eql operation #.*OP-CMP*)))
 ;; CMP does not allow a LOCK prefix.
 (!!ms-fresh :lock-prefix prefixes))

(p2 (prefixes-slice :group-2-prefix prefixes))
(byte-operand? (eql 0 (the (unsigned-byte 1)
                           (logand 1 opcode))))
((the (integer 1 8) operand-size)
 (select-operand-size byte-operand? rex-byte nil prefixes))

(G (rgfi-size operand-size
              (the (unsigned-byte 4)
                (reg-index reg rex-byte #.*r*))
              rex-byte x86))

(p4? (eql #.*addr-size-override*
          (prefixes-slice :group-4-prefix prefixes)))

(inst-ac? t)
((mv flg0 E (the (unsigned-byte 3) increment-RIP-by)
     (the (signed-byte #.*max-linear-address-size*) E-addr)
     x86)
 (x86-operand-from-modr/m-and-sib-bytes
  #.*rgf-access* operand-size inst-ac?
  nil ;; Not a memory pointer operand
  p2 p4? temp-rip rex-byte r/m mod sib
  0 ;; No immediate operand
  x86))
((when flg0)
 (!!ms-fresh :x86-operand-from-modr/m-and-sib-bytes flg0))

((the (signed-byte #.*max-linear-address-size+1*) temp-rip)
 (+ temp-rip increment-RIP-by))
((when (mbe :logic (not (canonical-address-p temp-rip))
```

```
            :exec (<= #.*2^47*
                      (the (signed-byte
                            #.*max-linear-address-size+1*)
                        temp-rip))))
 (!!ms-fresh :temp-rip-not-canonical temp-rip))
((the (signed-byte #.*max-linear-address-size+1*) addr-diff)
 (-
  (the (signed-byte #.*max-linear-address-size*)
    temp-rip)
  (the (signed-byte #.*max-linear-address-size*)
    start-rip)))
((when (< 15 addr-diff))
 (!!ms-fresh :instruction-length addr-diff))

;; Everything above this point is just further decoding
;; the instruction and fetching its operands.

;; Operation Specification:

;; Computing the flags and the result:
((the (unsigned-byte 32) input-rflags) (rflags x86))
((mv result
     (the (unsigned-byte 32) output-rflags)
     (the (unsigned-byte 32) undefined-flags))
 (gpr-arith/logic-spec
  operand-size operation E G input-rflags))

;; Updating the x86 state with the result and eflags.
((mv flg1 x86)
 (if (or (eql operation #.*OP-CMP*)
         (eql operation #.*OP-TEST*))
     ;; CMP and TEST modify just the flags.
     (mv nil x86)
   (x86-operand-to-reg/mem
    operand-size inst-ac?
    nil ;; Not a memory pointer operand
    result
```

```
        (the (signed-byte #.*max-linear-address-size*) E-addr)
        rex-byte r/m mod x86)))
    ((when flg1)
     (!!ms-fresh :x86-operand-to-reg/mem flg1))

    (x86 (write-user-rflags output-rflags undefined-flags x86))
    (x86 (!rip temp-rip x86)))

x86))
```

Appendix C

x86 Step Function

The step function, `x86-fetch-decode-execute`, was introduced in Chapter 3 and discussed in some detail in Chapter 8. Here, we present its ACL2 definition.

```
(defun x86-fetch-decode-execute (x86)
  ;; Guards elided.
  (b* ((ctx 'x86-fetch-decode-execute)
       ;; We don't want our interpreter to take a step if the
       ;; machine is in a bad state.  Such checks are made in
       ;; x86-run but I am duplicating them here in case this
       ;; function is being used at the top-level.
       ((when (or (ms x86) (fault x86))) x86)

       (start-rip (the (signed-byte #.*max-linear-address-size*)
                    (rip x86)))

       ((mv flg0 (the (unsigned-byte 44) prefixes) x86)
        (get-prefixes start-rip 0 15 x86))
       ;; Among other errors (including if there are 15 prefix
       ;; bytes, which leaves no room for an opcode byte in a
       ;; legal instruction), if get-prefixes detects a
       ;; non-canonical address while attempting to fetch
       ;; prefixes, flg0 will be non-nil.
       ((when flg0)
        (!!ms-fresh :error-in-reading-prefixes flg0))

       ((the (unsigned-byte 8) opcode/rex/escape-byte)
        (prefixes-slice :next-byte prefixes))
```

```
((the (unsigned-byte 4) prefix-length)
 (prefixes-slice :num-prefixes prefixes))
((the (signed-byte 49) temp-rip)
 (if (equal 0 prefix-length)
     (+ 1 start-rip)
   (+ 1 prefix-length start-rip)))

((when (mbe
        :logic (not (canonical-address-p temp-rip))
        :exec (<= #.*2^47*
                  (the (signed-byte
                        #.*max-linear-address-size+1*)
                    temp-rip))))
 (!!ms-fresh :non-canonical-address-encountered temp-rip))

;; If opcode/rex/escape-byte is a rex byte, it is filed
;; away in rex-byte.
((the (unsigned-byte 8) rex-byte)
 (if (and ;; 64-bit-mode
      (equal (the (unsigned-byte 4)
               (ash opcode/rex/escape-byte -4))
             4))
     opcode/rex/escape-byte
   0))

((mv flg1 (the (unsigned-byte 8) opcode/escape-byte) x86)
 (if (equal 0 rex-byte)
     (mv nil opcode/rex/escape-byte x86)
   (rm08 temp-rip :x x86)))
((when flg1)
 (!!ms-fresh :opcode/escape-byte-read-error flg1))

((mv flg2 (the (signed-byte 49) temp-rip))
 (if (equal rex-byte 0)
     ;; We know temp-rip is canonical from the previous
     ;; check.
```

```
      (mv nil temp-rip)
    (let
        ((temp-rip
          (the (signed-byte #.*max-linear-address-size+1*)
            (1+ temp-rip))))
      ;; We need to check whether (1+ temp-rip) is
      ;; canonical or not.
      (if (mbe
           :logic (canonical-address-p temp-rip)
           :exec (< (the (signed-byte
                          #.*max-linear-address-size+1*)
                      temp-rip)
                    #.*2^47*))
          (mv nil temp-rip)
        (mv t temp-rip)))))

((when flg2)
 (!!ms-fresh :non-canonical-address-encountered temp-rip))

;; Possible values of opcode/escape-byte:

;; 1. An opcode of the primary opcode map: this function
;;    prefetches the ModR/M and SIB bytes for these
;;    opcodes.  The function "top-level-opcode-execute"
;;    case-splits on this byte and calls the appropriate
;;    step function.

;; 2. #x0F -- two-byte opcode indicator: modr/m? is set to
;;    NIL (see *onebyte-has-modrm-lst* in constants.lisp).
;;    No ModR/M and SIB bytes are prefetched by this
;;    function for the two-byte opcode map.  Inside
;;    "top-level-opcode-execute", we call
;;    "two-byte-opcode-decode-and-execute", where we fetch
;;    the ModR/M and SIB bytes for these opcodes.

;; 3. #x8F: Depending on the value of ModR/M.reg,
;;    "top-level-opcode-execute" either calls the one-byte
```

```
;;    POP instruction or escapes to the XOP opcode map.

;; 4. #xC4, #xC5: Escape to the VEX opcode map.  Note that
;;    in this case, the ModR/M and SIB bytes will be
;;    prefetched by this function, and TEMP-RIP will be
;;    incremented accordingly.

;; The opcode/escape-byte should not contain any of the
;; prefix bytes -- by this point, all prefix bytes are
;; processed.

;; Note that modr/m? will be nil for #x0F and temp-rip
;; will not be incremented beyond this point in this
;; function for two-byte opcodes.

;; The modr/m and sib byte prefetching in this function is
;; "biased" towards the primary opcode map.
;; two-byte-opcode-decode-and-execute does its own
;; prefetching.  We made this choice to take advantage of
;; the fact that the most frequently encountered
;; instructions are from the primary opcode map.  Another
;; reason is that the instruction encoding syntax is
;; clearer to understand; this is a nice way of seeing how
;; one opcode map escapes into the other.

(modr/m? (x86-one-byte-opcode-ModR/M-p opcode/escape-byte))
((mv flg3 (the (unsigned-byte 8) modr/m) x86)
 (if modr/m?
     (rm08 temp-rip :x x86)
   (mv nil 0 x86)))
((when flg3)
 (!!ms-fresh :modr/m-byte-read-error flg2))

((mv flg4 (the (signed-byte 49) temp-rip))
 (if modr/m?
     (let
         ((temp-rip
```

```
            (the (signed-byte #.*max-linear-address-size+1*)
              (1+ temp-rip))))
        ;; We need to check whether (1+ temp-rip) is
        ;; canonical or not.
        (if (mbe
             :logic (canonical-address-p temp-rip)
             :exec (< (the (signed-byte
                            #.*max-linear-address-size+1*)
                        temp-rip)
                      #.*2^47*))
            (mv nil temp-rip)
          (mv t temp-rip)))
   ;; We know from the previous check that temp-rip is
   ;; canonical.
   (mv nil temp-rip)))

((when flg4)
 (!!ms-fresh :non-canonical-address-encountered temp-rip))

(sib? (and modr/m? (x86-decode-SIB-p modr/m)))

((mv flg5 (the (unsigned-byte 8) sib) x86)
 (if sib?
     (rm08 temp-rip :x x86)
   (mv nil 0 x86)))
((when flg5)
 (!!ms-fresh :sib-byte-read-error flg3))

((mv flg6 (the (signed-byte 49) temp-rip))
 (if sib?
     (let
         ((temp-rip
           (the (signed-byte #.*max-linear-address-size+2*)
             (1+ temp-rip))))
       ;; We need to check whether (1+ temp-rip) is
       ;; canonical.
       (if (mbe
```

```
                :logic (canonical-address-p temp-rip)
                :exec (< (the (signed-byte
                               #.*max-linear-address-size+2*)
                           temp-rip)
                         #.*2^47*))
              (mv nil temp-rip)
            (mv t temp-rip)))
      ;; We know from the previous check that temp-rip is
      ;; canonical.
      (mv nil temp-rip)))

  ((when flg6)
   (!!ms-fresh :virtual-address-error temp-rip)))

(top-level-opcode-execute
 start-rip temp-rip prefixes rex-byte opcode/escape-byte
 modr/m sib x86)))
```

Appendix D

Controlling Symbolic Simulation of x86 Programs

In this chapter, we present some ACL2 events that enable efficient and automatic symbolic simulation of x86 instructions.

D.1 Normalizing x86 State Accesses and Updates

See Section 5.3 of this dissertation for a discussion of how the top-level interface functions such as `memi` and `!memi` help in providing both reasoning and execution efficiency — these are defined in terms of the universal accessors and updater functions `xr` and `xw` for reasoning and abstract stobj exports for execution.

```
(defun-inline memi (i x86)
  ;; Guards elided.
  (mbe :logic (xr :mem i x86)
       :exec (memi* i x86)))

(defun-inline !memi (i v x86)
  ;; Guards elided.
  (mbe :logic (xw :mem i v x86)
       :exec (!memi* i v x86)))

(defun xr (fld index x86)
  ;; Guards elided.
  (case fld
    (:rgf (rgfi* index x86))
```

```
    (:mem (memi* index x86))
    ;; Other cases elided...
    (:os-info (os-info* x86))
    (otherwise nil)))

(defun xw (fld index value x86)
  ;; Guards elided.
  (case fld
    (:rgf (!rgfi* index value x86))
    ;; Other cases elided...
    (:os-info (!os-info* value x86))
    (:mem (!memi* index value x86))
    (otherwise x86)))
```

D.2 Non-Interference and Other Similar Theorems

We introduce our rules that help in controlling symbolic simulation by describing the effects of read and write operations with symbolic values in Chapter 3 of this dissertation. Their ACL2 definitions are presented below.

```
;; Read-over-Write Theorems
(defthm xr-xw-intra-array-field
  (implies (member fld *x86-array-fields*)
           (equal (xr fld i (xw fld j v x86))
                  (if (equal i j)
                      v
                    (xr fld i x86)))))

(defthm xr-xw-intra-simple-field
  (implies (member fld *x86-simple-fields*)
           (equal (xr fld i (xw fld j v x86))
                  v)))

(defthm xr-xw-inter-field
```

```
  (implies (case-split (not (equal fld1 fld2)))
           (equal (xr fld2 i2 (xw fld1 i1 v x86))
                  (xr fld2 i2 x86))))

;; Write-over-Write Theorems
(defthm xw-xw-intra-array-field-shadow-writes
  (implies (member fld *x86-array-fields*)
           (equal (xw fld i v2 (xw fld i v1 x86))
                  (xw fld i v2 x86))))

(defthm xw-xw-intra-simple-field-shadow-writes
  (implies (member fld *x86-simple-fields*)
           (equal (xw fld i v2 (xw fld j v1 x86))
                  (xw fld i v2 x86))))

(defthm xw-xw-intra-field-arrange-writes
  (implies (and (member fld *x86-array-fields*)
                (not (equal i1 i2)))
           (equal (xw fld i2 v2 (xw fld i1 v1 x86))
                  (xw fld i1 v1 (xw fld i2 v2 x86)))))

(defthm xw-xw-inter-field-arrange-writes
  (implies (not (equal fld1 fld2))
           (equal (xw fld2 i2 v2 (xw fld1 i1 v1 x86))
                  (xw fld1 i1 v1 (xw fld2 i2 v2 x86)))))

;; Writing-the-Read Theorem
(defthm xw-xr
  (implies (and (equal v (xr fld i x86))
                (x86p x86))
           (equal (xw fld i v x86) x86)))


;; State Well-Formedness Theorem
(defthm x86p-xw
  (implies
   (and (member fld *x86-fields*)
```

```
      (case fld
        (:rgf          (and (integerp index)
                            (signed-byte-p 64 value)))

        ;; Other cases elided...

        (:os-info      (keywordp value))
        (:mem          (and (integerp index)
                            (unsigned-byte-p 8 value)))
        (otherwise     (equal index 0)))
      (x86p x86))
  (x86p (xw fld index value x86))))
```

D.3 Unwinding the x86 Interpreter

We introduce rules that control the unwinding of our x86 ISA interpreter in Chapter 9 of this dissertation. Here, we present the ACL2 definitions of these rules.

We omit the step function opener rule for the system-level marking mode. An interested reader can find that rule in our source code, available online [51].

```
(defthm x86-fetch-decode-execute-opener
  ;; This theorem is applicable to the user-level mode
  ;; and the system-level non-marking mode of operation.
    (implies
     (and
      (not (ms x86))
      (not (fault x86))
      (equal start-rip (rip x86))
      (not (mv-nth 0 (get-prefixes start-rip 0 15 x86)))
      (equal prefixes (mv-nth 1 (get-prefixes start-rip 0 15 x86)))
      (equal opcode/rex/escape-byte
             (prefixes-slice :next-byte prefixes))
      (equal prefix-length (prefixes-slice :num-prefixes prefixes))
      (equal temp-rip0 (if (equal prefix-length 0)
```

```
                    (+ 1 start-rip)
                   (+ prefix-length start-rip 1)))
(equal rex-byte (if (equal (ash opcode/rex/escape-byte -4) 4)
                    opcode/rex/escape-byte
                  0))
(equal opcode/escape-byte
       (if (equal rex-byte 0)
           opcode/rex/escape-byte
         (mv-nth 1 (rm08 temp-rip0 :x x86))))
(equal temp-rip1 (if (equal rex-byte 0)
                     temp-rip0 (1+ temp-rip0)))
(equal modr/m?
       (x86-one-byte-opcode-modr/m-p opcode/escape-byte))
(equal modr/m (if modr/m?
                  (mv-nth 1 (rm08 temp-rip1 :x x86))
                0))
(equal temp-rip2 (if modr/m? (1+ temp-rip1) temp-rip1))
(equal sib? (and modr/m? (x86-decode-sib-p modr/m)))
(equal sib (if sib? (mv-nth 1 (rm08 temp-rip2 :x x86)) 0))
(equal temp-rip3 (if sib? (1+ temp-rip2) temp-rip2))

(or (user-level-mode x86)
    (and (not (user-level-mode x86))
         (not (page-structure-marking-mode x86))))
(canonical-address-p temp-rip0)
(if (and (equal prefix-length 0)
         (equal rex-byte 0)
         (not modr/m?))
    ;; One byte instruction --- all we need to know is that
    ;; the new RIP is canonical, not that there's no error
    ;; in reading a value from that address.
    t
  (not (mv-nth 0 (rm08 temp-rip0 :x x86))))
(if (equal rex-byte 0)
    t
  (canonical-address-p temp-rip1))
(if modr/m?
```

```
          (and (canonical-address-p temp-rip2)
               (not (mv-nth 0 (rm08 temp-rip1 :x x86))))
        t)
      (if sib?
          (and (canonical-address-p temp-rip3)
               (not (mv-nth 0 (rm08 temp-rip2 :x x86))))
        t)
      (x86p x86))
     (equal (x86-fetch-decode-execute x86)
            (top-level-opcode-execute
             start-rip temp-rip3 prefixes rex-byte
             opcode/escape-byte modr/m sib x86))))

;; Run Function Opener Rules
(defthm x86-run-halted
  (implies (or (ms x86) (fault x86))
           (equal (x86-run n x86) x86)))

(defthm x86-run-opener-not-ms-not-fault-zp-n
  (implies (and (syntaxp (quotep n))
                (zp n))
           (equal (x86-run n x86) x86)))

(defthm x86-run-opener-not-ms-not-zp-n
  (implies (and (not (ms x86))
                (not (fault x86))
                (syntaxp (quotep n))
                (not (zp n)))
           (equal (x86-run n x86)
                  (x86-run (1- n)
                           (x86-fetch-decode-execute x86)))))

;; Run Function Sequential Composition Rule
(defthm x86-run-plus
  (implies (and (natp n1)
                (natp n2)
                (syntaxp (quotep n1)))
```

```
(equal (x86-run (clk+ n1 n2) x86)
       (x86-run n2 (x86-run n1 x86))))
```

Appendix E

Zero-Copy Program

We present the verification of the zero-copy program in Chapter 12 of this dissertation. The C source of this program is presented below.

```
#define CR3_PDB_SHIFT 12

typedef unsigned long long u64;

#define _direct_map(x) (x);

u64 rewire_dst_to_src (u64 src_la, u64 dst_la) {

  u64 cr3;
  u64 pml4e_src_pa, pdpte_src_pa, src_pa;
  u64 pml4e_dst_pa, pdpte_dst_pa, modified_dst_pa;
  u64 copy_pdpte_ret_stat;

  __asm__ __volatile__
    ( /* Get cr3. */
     "mov %%cr3, %%rax\n\t"
     "mov %%rax, %0\n\t"
     : "=m"(cr3)
     :
     : "%rax"
      );

  /* Obtain PDPTE address for src_la. */
  pml4e_src_pa   = pml4e_paddr(cr3, src_la);
  pdpte_src_pa   = pdpte_paddr(pml4e_src_pa, src_la);
```

```
  if (pdpte_src_pa == -1) return -1;
  src_pa         = paddr(pdpte_src_pa, src_la);
  if (src_pa       == -1) return -1;

  /* Obtain PDPTE address for dst_la. */
  pml4e_dst_pa   = pml4e_paddr(cr3, dst_la);
  pdpte_dst_pa   = pdpte_paddr(pml4e_dst_pa, dst_la);
  if (pdpte_dst_pa == -1) return -1;

  /* Map dst_la to src_pa. */
  copy_pdpte_ret_stat = copy_pdpte(pdpte_src_pa, pdpte_dst_pa);
  if (copy_pdpte_ret_stat == -1) return -1;

  /* Get physical address corresponding to dst_la. */
  modified_dst_pa = paddr(pdpte_dst_pa, dst_la);
  if (modified_dst_pa       == -1) return -1;

  if (modified_dst_pa == src_pa)
    return 1; /* Success */
  return 0;   /* Failure */

}

u64 part_select (u64 x, u64 low, u64 high) {

  u64 width, val, mask;
  width = high - low + 1;
  mask = (1UL << width) - 1;
  val =  mask & (x >> low);
  return (val);

}

u64 part_install (u64 val, u64 x, u64 low, u64 high) {

  u64 width, mask, ret;
  width = high - low + 1;
```

```
  mask = (1UL << width) - 1;
  ret = (((~(mask << low)) & x) | (val << low));
  return (ret);

}

u64 pml4e_paddr (u64 cr3, u64 vaddr) {
  /* Input: Contents of the CR3 register and the virtual address
     Output: Physical address of the entry in PML4 table that
     corresponds to vaddr */

  u64 pml4_table_base_paddr;
  u64 paddr;

  pml4_table_base_paddr = \
    _direct_map((cr3 >> CR3_PDB_SHIFT) << CR3_PDB_SHIFT);

  /* Address of PML4E:
     Bits 51:12 are from CR3.
     Bits 11:3 are bits 47:39 of vaddr.
     Bits 2:0 are 0. */
  paddr = part_install (part_select (vaddr, 39, 47),
                        pml4_table_base_paddr, 3, 11);
  return (paddr);

}

u64 pdpte_paddr (u64 pml4e_paddr, u64 vaddr) {
  /* Input: Physical address of the PML4E and the virtual address
     Output: Physical address of the entry in PDPT table that
     corresponds to vaddr */

  u64 pdpt_table_base_addr, pml4e, paddr;

  /* Read the PML4E entry from pml4e_paddr: */
  pml4e = *((u64 *)pml4e_paddr);
  /* Return error if the PML4E has the P bit cleared. */
```

```
  if ((pml4e & 1) == 0) {
    return -1;
  }

  pdpt_table_base_addr = \
    _direct_map(part_select(pml4e, 12, 51) << 12);

  /* Address of PDPTE:
     Bits 51:12 are from the PML4E.
     Bits 11:3 are bits 38:30 of vaddr.
     Bits 2:0 are 0.  */

  paddr = part_install (part_select (vaddr, 30, 38),
                        pdpt_table_base_addr, 3, 11);
  return (paddr);

}

u64 paddr (u64 pdpte_addr, u64 vaddr) {
  /* Input: Physical address of the PDPTE and the virtual address
     Output: Physical address corresponding to vaddr */

  u64 page_base_paddr, pdpte, paddr;

  /* Read the PDPTE from the pte_addr: */
  pdpte = *((u64 *)pdpte_addr);
  /* Return error if the PDPTE has the P or PS bit cleared. */
  if (((pdpte & 1) == 0) || (part_select(pdpte, 7, 7) == 0)) {
    return -1;
  }

  page_base_paddr = _direct_map(part_select(pdpte, 30, 51) << 30);

  /* Physical Address corresponding to vaddr:
     Bits 51:30 are from the PDPTE.
     Bits 29:0 are bits 29:0 of vaddr. */
```

```
  paddr = part_install (part_select (vaddr, 0, 29),
                        page_base_paddr, 0, 29);

  return (paddr);

}

u64 copy_pdpte (u64 src_pdpte_paddr, u64 dst_pdpte_paddr) {
  // Input: Physical addresses of the PDPTE for the source and
  // destination
  // Output: -1 if error, 0 otherwise

    u64 src_page_base_paddr_field, src_pdpte;
    u64 dst_pdpte, modified_dst_pdpte;

  /* Read the PDPTE from the src_pdpte_paddr: */
  src_pdpte = *((u64 *)src_pdpte_paddr);
  /* Return error if the PDPTE has the P or PS bit cleared. */
  if (((src_pdpte & 1) == 0) ||
      (part_select(src_pdpte, 7, 7) == 0)) {
    return -1;
  }

  src_page_base_paddr_field = part_select(src_pdpte, 30, 51);

  /* Write src_page_base_paddr_field to the dst PDPTE. */
  dst_pdpte = *((u64 *)dst_pdpte_paddr);
  modified_dst_pdpte = \
    part_install(src_page_base_paddr_field, dst_pdpte, 30, 51);
  *((u64 *)dst_pdpte_paddr) = modified_dst_pdpte;

  return 0;
}
```

Bibliography

[1] ACL2 Book: Arithmetic. Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____ARITHMETIC-1. *[Cited on page 41]*

[2] ACL2 Book: `b*` Macro. Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____B_A2. *[Cited on pages 43 and 164]*

[3] ACL2 Books: Codewalker. Online; accessed: October 2016. https://github.com/acl2/acl2/tree/master/books/projects/codewalker. *[Cited on pages 22 and 217]*

[4] ACL2 Books: GL. Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____GL. *[Cited on page 41]*

[5] ACL2 Books: Proof Automation. Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____PROOF-AUTOMATION. *[Cited on page 41]*

[6] ACL2 Books: RTL. Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____RTL. *[Cited on page 41]*

[7] ACL2 Books: Standard (STD). Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____STD. *[Cited on page 41]*

[8] ACL2 Books: `bitops` – A Library for Reasoning about Bit-Vector Arithmetic. Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____BITOPS. *[Cited on page 41]*

[9] ACL2 Books: y86 Specifications. Online; accessed: October 2016. https://github.com/acl2/acl2/tree/master/books/models/y86. *[Cited on page 19]*

[10] ACL2 Documentation. Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/current/combined-manual/?topic=ACL2____TOP. *[Cited on page 41]*

[11] ACL2 Documentation: Congruences. Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____CONGRUENCE. *[Cited on pages 132 and 150]*

[12] ACL2 Documentation: Free Variables. Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____FREE-VARIABLES. *[Cited on page 152]*

[13] ACL2 Documentation: Guards. Online; accessed: October 2016.
http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____GUARD. *[Cited on page 45]*

[14] ACL2 Documentation: `break-rewrite`. Online; accessed: October 2016.
http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____BREAK-REWRITE. *[Cited on page 125]*

[15] ACL2 Documentation: `defun-notinline`. Online; accessed: October 2016.
http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____DEFUN-NOTINLINE. *[Cited on pages 96 and 97]*

[16] ACL2 Documentation: `remove-hyps`. Online; accessed: October 2016.
http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____REMOVE-HYPS. *[Cited on page 129]*

[17] ACL2 Documentation: Untouchable Functions. Online; accessed: October 2016.
http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html?topic=ACL2____PUSH-UNTOUCHABLE. *[Cited on page 92]*

[18] ACL2: Github Page. Online; accessed: October 2016.
https://github.com/acl2/acl2. *[Cited on page 41]*

[19] ACL2 Home Page. Online; accessed: October 2016.
http://www.cs.utexas.edu/users/moore/acl2. *[Cited on pages 2 and 28]*

[20] ACL2: Interesting Applications. Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2____INTERESTING-APPLICATIONS. *[Cited on page 40]*

[21] AMD Developer Guides and Manuals. Online; accessed: October 2016. http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/. *[Cited on page 7]*

[22] CCL Manual: Code Coverage. Online; accessed: October 2016. http://ccl.clozure.com/manual/chapter4.13.html. *[Cited on page 116]*

[23] CCL Manual: Foreign Function Interface. Online; accessed: October 2016. http://ccl.clozure.com/manual/chapter13.html. *[Cited on page 100]*

[24] Centaur Technology. Online; accessed: October 2016. http://www.centtech.com. *[Cited on page 217]*

[25] Clang Static Analyzer. Online; accessed: October 2016. http://clang-analyzer.llvm.org/. *[Cited on page 19]*

[26] CLHS (Common Lisp HyperSpec). Online; accessed: October 2016. http://www.lispworks.com/reference/HyperSpec/index.html. *[Cited on page 44]*

[27] CLHS: Integer. Online; accessed: October 2016. http://www.lispworks.com/documentation/HyperSpec/Body/t_intege.htm. *[Cited on page 46]*

[28] CLHS: `logcount` Function. Online; accessed: October 2016. http://www.lispworks.com/documentation/HyperSpec/Body/f_logcou.htm#logcount. *[Cited on page 163]*

[29] CompCert. Online; accessed: October 2016. http://compcert.inria.fr. *[Cited on page 20]*

[30] Coq Proof Assistant: Home Page. Online; accessed: October 2016. http://coq.inria.fr/. *[Cited on pages 18 and 21]*

[31] Coverity Static Analysis. Online; accessed: October 2016. http://www.coverity.com. *[Cited on page 19]*

[32] GCC, the GNU Compiler Collection. Online; accessed: October 2016. https://gcc.gnu.org/. *[Cited on pages 6 and 32]*

[33] GDB: The GNU Project Debugger. Online; accessed: October 2016. http://www.gnu.org/software/gdb/. *[Cited on page 114]*

[34] GL Term-Level Reasoning. Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=GL____TERM-LEVEL-REASONING. *[Cited on page 161]*

[35] Grammatech Static Analysis. Online; accessed: October 2016. http://www.grammatech.com/. *[Cited on page 19]*

[36] Intel 64 and IA-32 Architectures Software Developer's Manual: Section 2.2.10: Intel 64 Architecture, Vol. 1. Online. Order Number: 325462-059US. (June

2016).
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. *[Cited on page 6]*

[37] Intel 64 and IA-32 Architectures Software Developer's Manual: Section 3.2.4 Segmentation in IA-32e Mode, Vol. 3A. Online. Order Number: 325462-059US. (June 2016).
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. *[Cited on page 84]*

[38] Intel 64 and IA-32 Architectures Software Developer's Manual: Section 3.3.7.1 Canonical Addressing, Vol. 1. Online. Order Number: 325462-059US. (June 2016).
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. *[Cited on page 56]*

[39] Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 2.: Appendix A. Online. Order Number: 325462-059US. (June 2016).
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. *[Cited on page 109]*

[40] Intel 64 and IA-32 Architectures Software Developer's Manuals. Online. Order Number: 325462-059US. (June 2016).
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. *[Cited on pages 7 and 107]*

[41] Intel Xeon Processor E3-1200 v3 Product Family (Specification Update). Online. Reference Number: 328908-015US. (April 2016). http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf. *[Cited on page 115]*

[42] Kestrel Technology. Online; accessed: October 2016. http://kestreltechnology.com/. *[Cited on page 218]*

[43] Memory Management: Linux System Administrators Guide. Online; accessed: October 2016. http://www.tldp.org/LDP/sag/html/memory-management.html. *[Cited on page 20]*

[44] Memory Management: Windows System Administrators Guide. Online; accessed: October 2016. http://msdn.microsoft.com/en-us/library/windows/desktop/aa366779(v=vs.85).aspx. *[Cited on page 20]*

[45] sel4: General Dynamics C4 Systems. Online; accessed: October 2016. http://sel4.systems/. *[Cited on pages 23 and 40]*

[46] The Coq Proof Assistant: Reference Manual (Version 8.5pl2). Online; accessed: October 2016. https://coq.inria.fr/distrib/current/refman/. *[Cited on page 18]*

[47] The LLVM Compiler Infrastructure. Online; accessed: October 2016. http://llvm.org/. *[Cited on pages 6 and 32]*

[48] The Pi Verifying Compiler. Online; accessed: October 2016. http://theory.stanford.edu/~arbrad/pivc/. *[Cited on page 20]*

[49] `x86isa`: Code Proof Debugging. Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=X86ISA____X86-PROGRAMS-PROOF-DEBUGGING. *[Cited on page 127]*

[50] `x86isa`: Documentation. Online; accessed: October 2016. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____X86ISA. *[Cited on pages 10 and 213]*

[51] `x86isa` Library in the ACL2 Community Books Project on Github. Online; accessed: October 2016. https://github.com/acl2/acl2/tree/master/books/projects/x86isa. *[Cited on pages 10, 213, 228, and 241]*

[52] `x86isa`: Some Test Cases for Decoding (A Log Of Instruction Decoding Bugs Found By Performing Co-Simulations Against QEMU). Online; accessed: October 2016. https://github.com/acl2/acl2/tree/master/books/projects/x86isa/tools/execution/examples/documenting-edge-cases. *[Cited on page 115]*

[53] `x86isa`: Verifying Pop-Count with a Symbolic Initial x86 State. Online; accessed: October 2016. https://github.com/acl2/acl2/tree/master/books/projects/x86isa/proofs/popcount/popcount-general.lisp. *[Cited on page 182]*

[54] Veracode: White Box and Black Box Testing (SAST and DAST). Online; accessed: October 2016. http://www.veracode.com/. *[Cited on page 19]*

[55] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-End Verification of Processors with ISA-Formal. In *International Conference on Computer Aided Verification*, pages 42–58. Springer, 2016. *[Cited on page 15]*

[56] Eyad Alkassar, Ernie Cohen, Mark Hillebrand, Mikhail Kovalev, and Wolfgang J. Paul. Verifying Shadow Page Table Algorithms. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 267–270, Oct 2010. *[Cited on page 23]*

[57] Eyad Alkassar, Wolfgang J. Paul, Artem Starostin, and Alexandra Tsyban. Pervasive Verification of an OS Microkernel. In *International Conference on Verified Software: Theories, Tools, and Experiments*, pages 71–85. Springer, 2010. *[Cited on page 23]*

[58] Anna Slobodova, Jared Davis, Sol Swords, and Warren A. Hunt, Jr. A Flexible Formal Verification Framework for Industrial Scale Validation. In

9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2011, pages 89–97, July 2011. *[Cited on page 166]*

[59] Anthony Fox. Formal Specification and Verification of ARM6. In *Theorem Proving in Higher Order Logics*, pages 25–40. Springer, 2003. *[Cited on page 18]*

[60] Anthony Fox. Directions in ISA Specification. *Interactive Theorem Proving (ITP)*, pages 338–344, 2012. *[Cited on page 17]*

[61] Anthony Fox and Magnus O. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In Matt Kaufmann and Lawrence C. Paulson, editor, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer Berlin Heidelberg, 2010. *[Cited on page 18]*

[62] Arthur D. Flatau. *A Verified Implementation Of An Applicative Language With Dynamic Storage Allocation*. PhD thesis, The University of Texas at Austin, 1992. *[Cited on page 20]*

[63] Zachry Basnight, Jonathan Butts, Juan Lopez, and Thomas Dube. Firmware Modification Attacks on Programmable Logic Controllers. *International Journal of Critical Infrastructure Protection*, 6(2):76–84, 2013. *[Cited on page 217]*

[64] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, 53(2):66–75, February 2010. *[Cited on page 19]*

[65] William R. Bevier. *A Verified Operating System Kernel*. PhD thesis, Department of Computer Science, The University of Texas at Austin, 1987. *[Cited on page 21]*

[66] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all Together–Formal Verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4-5):411–430, 2006. *[Cited on page 23]*

[67] Bishop Brock, Matt Kaufmann, and J S. Moore. Rewriting with Equivalence Relations in ACL2. *Journal of Automated Reasoning*, 40(4):293–306, 2008. *[Cited on pages 132 and 150]*

[68] Robert S Boyer, Matt Kaufmann, and J S. Moore. The Boyer-Moore Theorem Prover and its Interactive Enhancement. *Computers & Mathematics with Applications*, 29(2):27–62, 1995. *[Cited on page 16]*

[69] Brian Campbell and Ian Stark. Extracting Behaviour from an Executable Instruction Set Model. In Ruzica Piskac and Muralidhar Talupur, editors, *Formal Methods in Computer-Aided Design, (FMCAD 2016)*, pages 33–40, 2016. *[Cited on page 116]*

[70] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 2 edition, 1988. *[Cited on page 169]*

[71] Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. Verifying Com-

piler Transformations for Concurrent Programs. Technical Report MSR-TR-2008-171, Microsoft Research, November 2008. *[Cited on page 20]*

[72] C. A. R. Hoare. An Axiomatic Basis For Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969. *[Cited on page 33]*

[73] C.A.R. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *J. ACM*, 50(1):63–69, January 2003. *[Cited on page 20]*

[74] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005. *[Cited on page 114]*

[75] Claude Marché. Jessie: An Intermediate Language For Java And C Verification. In *Proceedings of the 2007 workshop on Programming Languages meets Program Verification*, pages 1–2. ACM, 2007. *[Cited on page 19]*

[76] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012. *[Cited on page 19]*

[77] Paul Curzon. Deriving Correctness Properties of Compiled Code. *Formal Methods in System Design*, 3(1-2):83–115, 1993. *[Cited on page 20]*

[78] D. L. Clutterbuck and B. A. Carré. The Verification of Low-level Code. *Software Engineering Journal*, 3(3):97–111, 1988. *[Cited on page 21]*

[79] David A. Greve. Address Enumeration And Reasoning Over Linear Address Spaces. In *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004)*. *[Cited on pages 25 and 155]*

[80] David A. Greve. Scalable Normalization for Heap Manipulating Functions. In *Seventh International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2007)*. *[Cited on page 71]*

[81] David A. Greve. Symbolic Simulation of the JEM1 Microprocessor. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, pages 321–333. Springer Berlin Heidelberg, 1998. *[Cited on page 15]*

[82] David A. Greve and Matt Wilding. Dynamic Datastructures in ACL2: A Challenge. Online; accessed: October 2016. http://hokiepokie.org/docs/festival02.txt. *[Cited on page 24]*

[83] David A. Greve, Matthew Wilding, and David Hardin. High-Speed, Analyzable Simulators. In *Computer-Aided Reasoning*, pages 113–135. Springer, 2000. *[Cited on page 17]*

[84] David Cyrluk. Microprocessor Verification in PVS: A Methodology And Simple Example. Online; published: (February 1994). Accessed: October 2016. http://www.csl.sri.com/papers/csl-93-12/. *[Cited on page 15]*

[85] David L Rager, Jo Ebergen, Austin Lee, Dmitry Nadezhin, Ben Selfridge, and Cuong K Chau. A Brief Introduction to Oracle's Use of ACL2 in Veri-

fying Floating-point and Integer Arithmetic. In Matt Kaufmann and David L. Rager, editor, *Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, Texas, USA, 1-2 October 2015*, volume 192 of *Electronic Proceedings in Theoretical Computer Science*. Open Publishing Association, 2015. *[Cited on page 166]*

[86] Jared Davis and Matt Kaufmann. Industrial-Strength Documentation for ACL2. In *Proceedings of the 12th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2014, Vienna, Austria, July 12-13, 2014.*, pages 9–25, 2014. *[Cited on page 41]*

[87] Jared Davis, Anna Slobodova, and Sol Swords. Microcode Verification – Another Piece of the Microprocessor Verification Puzzle. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, volume 8558 of *Lecture Notes in Computer Science*, pages 1–16. Springer International Publishing, 2014. *[Cited on pages 166 and 217]*

[88] Eric Smith, Serita Nelesen, David Greve, and Matthew Wilding, and Raymond Richards. An ACL2 Library For Bags (Multisets). In *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004)*. *[Cited on page 25]*

[89] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture*

Notes in Computer Science, pages 23–42. Springer, 2009. *[Cited on pages 19 and 23]*

[90] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005. *[Cited on pages 7, 26, and 115]*

[91] Anthony Fox. Improved Tool Support for Machine-Code Decompilation in HOL4. In *International Conference on Interactive Theorem Proving*, pages 187–202. Springer, 2015. *[Cited on pages 17 and 18]*

[92] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, and others. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*, pages 207–220. ACM, 2009. *[Cited on pages 23 and 40]*

[93] Mike J. C. Gordon and Tom F. Melham, editors. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993. *[Cited on page 18]*

[94] Greg Morrisett. Scalable Formal Machine Models. In *Proceedings of the Second International Conference on Certified Programs and Proofs*, CPP'12, pages 1–3, Berlin, Heidelberg, 2012. Springer-Verlag. *[Cited on page 18]*

[95] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, Faster, Stronger SFI for the x86. In *Proceedings*

of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, pages 395–404. ACM, 2012. *[Cited on page 22]*

[96] David A. Greve, Matt Kaufmann, Panagiotis Manolios, J S. Moore, Sandip Ray, José-Luis Ruiz-Reina, Rob Sumners, Daron Vroon, and Matthew Wilding. Efficient Execution in an Automated Reasoning Environment. *J. Funct. Program.*, 18(1), 2008. *[Cited on page 46]*

[97] Hanbing Liu. A Solution to the Rockwell Challenge. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)*. *[Cited on page 25]*

[98] Hanbing Liu and J S. Moore. Java program verification via a JVM deep embedding in ACL2. In *International Conference on Theorem Proving in Higher Order Logics*, pages 184–200. Springer, 2004. *[Cited on page 17]*

[99] Claude Helmstetter, Vania Joloboff, and Hui Xiao. SimSoC: A Full System Simulation Software for Embedded Systems. In *Open-source Software for Scientific Computation (OSSC), 2009 IEEE International Workshop on*, pages 49–55. IEEE, 2009. *[Cited on page 18]*

[100] James Hendricks and Leendert van Doorn. Secure Bootstrap is Not Enough: Shoring Up the Trusted Computing Base. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, New York, NY, USA, 2004. ACM. *[Cited on page 217]*

[101] Alex Horn, Michael Tautschnig, Celina Val, Lihao Liang, Tom Melham, Jim Grundy, and Daniel Kroening. Formal Co-Validation of Low-Level Hardware/-Software Interfaces. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 121–128. IEEE, 2013. *[Cited on page 217]*

[102] J S. Moore. Mechanized Operational Semantics. Online; accessed: October 2016. Lectures in the Marktoberdorf Summer School (August 5-16, 2008). http://www.cs.utexas.edu/users/moore/publications/talks/marktoberdorf-08/index.html. *[Cited on pages 17, 33, and 120]*

[103] J S. Moore. Memory Taggings And Dynamic Data Structures. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)*. *[Cited on page 24]*

[104] J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Automated Reasoning Series, Kluwer Academic Publishers, 1996. *[Cited on page 16]*

[105] J S. Moore. Symbolic Simulation: An ACL2 Approach. In *International Conference on Formal Methods in Computer-Aided Design*, pages 334–350. Springer, 1998. *[Cited on page 36]*

[106] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell and Francesco Zappa Nardelli. The Semantics of Power and ARM Multiprocessor Machine Code. In *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, pages 13–24. ACM, 2009. *[Cited on page 18]*

[107] Jared Davis. *A Self-Verifying Theorem Prover*. PhD thesis, Department of Computer Science, The University of Texas at Austin, 2009. *[Cited on page 22]*

[108] Jean-Louis Boulanger. *Static Analysis of Software: The Abstract Interpretation*. John Wiley & Sons, 2013. *[Cited on page 19]*

[109] Jifeng He, C. A. R. Hoare, Martin Fränzle, Markus Müller-Olm, Ernst-Rüdiger Olderog, Michael Schenke, Michael R. Hansen, Anders P. Ravn, and Hans Rischel. Provably Correct Systems. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 288–335. Springer, 1994. *[Cited on page 16]*

[110] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002. *[Cited on page 24]*

[111] John Matthews, J S. Moore, Sandip Ray, and Daron Vroon. Verification Condition Generation via Theorem Proving. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 2006. *[Cited on page 21]*

[112] John McCarthy. *Towards a Mathematical Science of Computation*, pages 35–56. Springer Netherlands, Dordrecht, 1993. *[Cited on page 33]*

[113] Jun Sawada and Warren A. Hunt, Jr. Verification of FM9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability. *Formal Methods in Systems Design*, 20(2):187–222, 2002. *[Cited on page 15]*

[114] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the Depths of C: Elaborating the de facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–15. ACM, 2016. *[Cited on page 20]*

[115] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, 1st edition, 2010. *[Cited on page 20]*

[116] Kevin P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux J.*, 1996(29es), September 1996. *[Cited on page 26]*

[117] Yousef A. Khalidi and Moti N. Thadani. An Efficient Zero-Copy I/O Framework for UNIX. Technical report, Mountain View, CA, USA, 1995. *[Cited on page 184]*

[118] Klaus Havelund and Thomas Pressburger. Model Checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000. *[Cited on page 19]*

[119] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems (TOCS)*, 32(1):2, 2014. *[Cited on page 23]*

[120] Kendra Kratkiewicz and Richard Lippmann. Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools. In *Proc. of Workshop on the Evaluation of Software Defect Detection Tools*, 2005. *[Cited on page 19]*

[121] Axel van Lamsweerde. Formal Specification: a Roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159. ACM, 2000. *[Cited on page 26]*

[122] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984. *[Cited on page 17]*

[123] Junghee Lim and Thomas Reps. TSL: A System for Generating Abstract Interpreters and its Application to Machine-code Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 35(1):4, 2013. *[Cited on page 22]*

[124] OS X ABI Mach-O File Format Reference. Online; accessed: October 2016. https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime/index.html. *[Cited on page 113]*

[125] Magnus O. Myreen. *Formal Verification of Machine-code Programs*. PhD thesis, University of Cambridge, Computer Laboratory, Trinity College., 2008. *[Cited on pages 21 and 217]*

[126] Magnus O. Myreen and Jared Davis. A Verified Runtime for a Verified Theorem Prover. *Interactive Theorem Proving*, pages 265–280, 2011. *[Cited on page 22]*

[127] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Decompilation Into Logic - Improved. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 78–81., 2012. *[Cited on page 21]*

[128] Marshall K. McKusick, George V. Neville-Neil, and Robert N. M. Watson. *Chapter 6: Memory Management, The Design and Implementation of the FreeBSD Operating System.* Addison Wesley Professional, 2014. *[Cited on page 20]*

[129] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System.* Addison-Wesley Professional, 2nd edition, 2014. *[Cited on page 99]*

[130] Matt Kaufmann and J S. Moore. Rough diamond: An extension of equivalence-based rewriting. In G. Klein and R. Gamboa, editors, *Proceedings of ITP 2014: 5th International Conference on Interactive Theorem Proving*, volume LNAI 8558, pages 537–542. Springer-Verlag, 2014. *[Cited on pages 132 and 150]*

[131] Matt Kaufmann and Rob Sumners. Efficient Rewriting of Data Structures in ACL2. In *Proceedings of 3rd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2002). [Cited on page 61]*

[132] Matt Kaufmann and Rob Sumners. Efficient Rewriting of Operations on Finite Structures in ACL2. In *ETAPS 2002: European Joint Conference on Theory and Practice of Software. Satellite workshop*, pages 141–150, 2002. *[Cited on page 69]*

[133] Matt Kaufmann, J S. Moore, Sandip Ray, and Erik Reeber. Integrating External Deduction Tools with {ACL2}. *Journal of Applied Logic*, 7(1):3 – 25, 2009. Special Issue: Empirically Successful Computerized Reasoning. *[Cited on pages 41 and 94]*

[134] Matt Kaufmann, Panagiotis Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, MA, June 2000. *[Cited on pages 2 and 28]*

[135] Matthew Wilding, David A. Greve, and David Hardin. Efficient Simulation Of Formal Processor Models. *Formal Methods in System Design*, 18(3):233–248, 2001. *[Cited on page 17]*

[136] John McCarthy. Correctness of a Compiler for Arithmetic Expressions. *Mathematical Aspects of Computer Science*, 1, 1967. *[Cited on page 20]*

[137] Michael Kerrisk. *The Linux Programming Interface*. No Starch Press, 2010. *[Cited on page 99]*

[138] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. Chapter 4: Object Files in System V Application Binary Interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99, 2005. *[Cited on page 113]*

[139] Michael Matz, Jan Hubiĉka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface. AMD64 Architecture Processor Supplement, 2013. *[Cited on page 104]*

[140] Mike Dahlin, Ryan Johnson, Robert B. Krug, Michael McCoyd, and William D. Young. Toward the Verification of a Simple Hypervisor. In *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2011, Austin, Texas, USA, November 3-4, 2011.*, pages 28–45, 2011. *[Cited on page 24]*

[141] F. L. Morris and C. B. Jones. An Early Program Proof by Alan Turing. 6(2):139–143, April/June 1984. *[Cited on page 278]*

[142] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):527–568, 1999. *[Cited on page 22]*

[143] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable Engineering of Real-World Semantics. In *ACM SIGPLAN Notices*, volume 49, pages 175–188. ACM, 2014. *[Cited on page 17]*

[144] Magnus O. Myreen, Mike Gordon, and Konrad Slind. Machine-Code Verification for Multiple Architectures - An Application of Decompilation into Logic. In *Formal Methods in Computer-Aided Design, 2008. FMCAD '08*, pages 1–8, Nov 2008. *[Cited on page 22]*

[145] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM. *[Cited on page 22]*

[146] George C. Necula and Peter Lee. Safe Kernel Extensions without Run-time Checking. *SIGOPS Operating Systems Review*, 30:229–244, 1996. *[Cited on page 22]*

[147] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. *[Cited on page 19]*

[148] Scott Owens, Susmit Sarkar, and Peter Sewell. A Better x86 Memory Model: x86-TSO. In *TPHOLs'09: Conference on Theorem Proving in Higher Order Logics, volume 5674 of LNCS*, pages 391–407. Springer, 2009. *[Cited on pages 18 and 216]*

[149] Sathish K. Palaniappan and Pramod B. Nagaraja. Efficient Data Transfer Through Zero Copy. *IBM developerworks*, 2008. *[Cited on page 184]*

[150] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010. *[Cited on pages 18 and 216]*

[151] Rafal Kolanski. *Verification of Programs in Virtual Memory using Separation Logic*. PhD thesis, UNSW, Sydney, Australia, July 2011. *[Cited on page 24]*

[152] Randal E. Bryant and David R. O'Hallaron. *Chapter 4: Processor Architecture,*

of Computer Systems: A Programmer's Perspective. Prentice-Hall, 2003. *[Cited on pages 19 and 24]*

[153] Alastair Reid. Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture. To Appear in Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design (FMCAD'16). *[Cited on page 15]*

[154] Richard J. Fateman. Comments on Factorial Programs. 2006. https://people.eecs.berkeley.edu/~fateman/papers/factorial.pdf. *[Cited on page 42]*

[155] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, and others. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 5). Technical Report UCAM-CL-TR-891, University of Cambridge, Computer Laboratory, 2016. *[Cited on page 17]*

[156] Robert S. Boyer and J S. Moore. Mechanized Formal Reasoning About Programs And Computing Machines. *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, pages 147–176, 1996. *[Cited on page 36]*

[157] Robert S. Boyer and J S. Moore. Single-threaded Objects in ACL2. In S. Krishnamurthy and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages (PADL)*, volume 2257 of *LNCS*, pages 9–27. Springer-Verlag, 2002. *[Cited on page 53]*

[158] Robert S. Boyer and Yuan Yu. Automated Proofs of Object Code for a Widely Used Microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996. *[Cited on pages 16 and 21]*

[159] Robert W. Floyd. Assigning Meanings to Programs. *Mathematical Aspects of Computer Science*, 19(19-32):1, 1967. *[Cited on page 33]*

[160] Roderick Chapman and Florian Schanda. Are We There Yet? 20 Years of Industrial Theorem Proving with SPARK. In Gerwin Klein and Ruben Gamboa, editor, *Interactive Theorem Proving*, volume 8558 of *Lecture Notes in Computer Science*, pages 17–26. Springer International Publishing, 2014. *[Cited on page 19]*

[161] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag. *[Cited on page 15]*

[162] Sandip Ray and J S. Moore. Proof Styles in Operational Semantics. In A. J. Hu and A. K. Martin, editors, *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *LNCS*, pages 67–81, Austin, TX, November 2004. Springer-Verlag. *[Cited on page 36]*

[163] Sandip Ray, Warren A. Hunt, Jr., John Matthews, and J S. Moore. A Me-

chanical Analysis of Program Verification Strategies. *Journal of Automated Reasoning*, 40(4):245–269, May 2008. *[Cited on pages 36 and 173]*

[164] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding Power Multiprocessors. *SIGPLAN Not.*, 46(6):175–186, June 2011. *[Cited on page 18]*

[165] Sean Anderson. Bit Twiddling Hacks. Online; accessed: October 2016. http://graphics.stanford.edu/~seander/bithacks.html. *[Cited on page 158]*

[166] Zhong Shao. Clean-Slate Development of Certified OS Kernels. In *Proceedings of the 2015 Workshop on Certified Programs and Proofs*, CPP '15, pages 95–96, New York, NY, USA, 2015. ACM. *[Cited on pages 18 and 23]*

[167] Shilpi Goel and Warren A. Hunt, Jr. Automated Code Proofs on a Formal Model of the x86. In *Verified Software: Theories, Tools, Experiments (VSTTE'13)*, volume 8164 of *Lecture Notes in Computer Science*, pages 222–241. Springer Berlin Heidelberg, 2014. *[Cited on pages 157 and 167]*

[168] Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann. Abstract Stobjs and Their Application to ISA Modeling. In *Proceedings of the ACL2 Workshop 2013, EPTCS 114, pp. 54-69*, 2013. *[Cited on page 65]*

[169] Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann. Engineering a Formal, Executable x86 ISA Simulator for Software Verification. In *Provably Correct Systems (ProCoS)*, 2015. *[Not cited]*

[170] Shilpi Goel, Warren A. Hunt, Jr., Matt Kaufmann, and Soumava Ghosh. Simulation and Formal Verification of x86 Machine-Code Programs That Make System Calls. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design (FMCAD'14)*, pages 18:91–98, 2014. *[Cited on page 157]*

[171] Axel Simon and Julian Kranz. The GDSL Toolkit: Generating Frontends for the Analysis of Machine Code. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14, pages 7:1–7:6, New York, NY, USA, 2014. ACM. *[Cited on page 17]*

[172] Sol Swords. *A Verified Framework for Symbolic Execution in the ACL2 Theorem Prover*. PhD thesis, Department of Computer Science, The University of Texas at Austin, 2010. *[Cited on page 160]*

[173] Dragan Stancevic. Zero Copy I: User-mode Perspective. *Linux J.*, 2003(105):3–, January 2003. *[Cited on page 184]*

[174] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus Myreen, and Jade Alglave. The Semantics of x86-CC Multiprocessor Machine Code. In *Proceedings of POPL 2009: the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 379–391, January 2009. *[Cited on page 18]*

[175] Sol Swords and Jared Davis. Bit-blasting ACL2 theorems. In *Proceedings of the 10th International Workshop on the ACL2 Theorem Prover and its Appli-*

cations, ACL2 2011, Austin, Texas, USA, November 3-4, 2011., pages 84–102, 2011. *[Cited on page 160]*

[176] Thomas Sewell, Magnus O. Myreen, and Gerwin Klein. Translation Validation for a Verified OS Kernel. In *ACM SIGPLAN Notices*, volume 48, pages 471–482. ACM, 2013. *[Cited on page 23]*

[177] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283. Springer Science & Business Media, 2002. *[Cited on page 18]*

[178] Alan M. Turing. Checking a Large Routine. pages 67–69, 1949. A corrected version is printed in [141]. The original is reprinted in [189, pp. 70–72]. *[Cited on page 21]*

[179] Ulan Degenbaev. *Formal Specification of the x86 Instruction Set Architecture*. PhD thesis, Universität des Saarlandes, 2012. *[Cited on page 17]*

[180] Warren A. Hunt, Jr. Microprocessor Design Verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989. *[Cited on page 16]*

[181] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *LNAI*. Springer-Verlag, 1994. *[Cited on page 15]*

[182] Warren A. Hunt, Jr. and Matt Kaufmann. A Formal Model of a Large Memory that Supports Efficient Execution. In *Proceedings of the 12th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2012, Cambrige, UK, October 22-25)*, 2012. *[Cited on page 56]*

[183] Warren A. Hunt, Jr., Matt Kaufmann, J S. Moore, and Anna Slobodova. Industrial Hardware and Software Verification with ACL2. In *Philosophical Transactions of the Royal Society (Article Number 20150399)*, volume 375, 2017 (to appear). *[Cited on page 40]*

[184] Warren A. Hunt, Jr., Sol Swords, Jared Davis, and Anna Slobodova. Use of Formal Verification at Centaur Technology. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 65–88. Springer, 2010. *[Cited on pages 15 and 166]*

[185] Nathan D. Wetzler. *Efficient, Mechanically-Verified Validation of Satisfiability Solvers*. PhD thesis, Department of Computer Science, The University of Texas at Austin, 2015. *[Cited on page 25]*

[186] Matthew M. Wilding, David A. Greve, Raymond J. Richards, and David S. Hardin. Formal Verification of Partition Management for the AAMP7G Microprocessor. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 175–191. Springer, 2010. *[Cited on page 15]*

[187] William D. Young. *A Verified Code Generator For A Subset Of Gypsy*. PhD thesis, The University of Texas at Austin, 1988. *[Cited on page 20]*

[188] William R. Bevier, Warren A. Hunt, Jr., J S. Moore, and William D. Young. Special Issue on System Verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989. *[Cited on pages 16, 40, and 173]*

[189] Michael R. Williams and Martin Campbell-Kelly, editors. *The Early British Computer Conferences*, volume 14 of *The Charles Babbage Institute reprint series for the History of Computing*. 1989. *[Cited on page 278]*

[190] Xavier Leroy. Formal Verification Of A Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009. *[Cited on page 20]*

[191] Xiaomu Shi. *Certification of an Instruction Set Simulator*. PhD thesis, Université de Grenoble, 2013. *[Cited on page 18]*

[192] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong. Certifying Low-Level Programs With Hardware Interrupts And Preemptive Threads. *Journal of Automated Reasoning*, 42(2):301–347, 2009. *[Cited on page 21]*

[193] William D. Young. A Mechanically Verified Code Generator. *Journal of Automated Reasoning*, 5(4):493–518, 1989. *[Cited on page 16]*

[194] Michael Zhivich, Tim Leek, and Richard Lippmann. Dynamic Buffer Overflow Detection. In *Workshop on the Evaluation of Software Defect Detection Tools*, volume 2005, 2005. *[Cited on page 19]*

[195] Zhong Shao and Bryan Ford. Advanced Development of Certified OS Kernels. Technical Report YALEU/DCS/TR-1436, Department of Computer Science, Yale University, New Haven, CT, July 2010.
http://flint.cs.yale.edu/publications/ctos.html. *[Cited on page 23]*